Maisterintutkielma

Tietojenkäsittelytiede

Jatkuva integraatio ja mallien yhteistoiminta koneoppimissovelluksissa

Toni Räsänen

4.5.2020

Matemaattis-luonnontieteellinen tiedekunta

Helsingin yliopisto

Ohjaaja(t)

Prof. Jukka K. Nurminen

Tarkastaja(t)

Prof. Jukka K. Nurminen, Prof. Tommi Mikkonen

Yhteystiedot

PL 68 (Pietari Kalmin katu 5)
00014 Helsingin yliopisto

Sähkopostiosoite: info@cs.helsinki.fi
URL: http://www.cs.helsinki.fi/

HELSINGIN YLIOPISTO – HELSINGFORS UNIVERSITET – UNIVERSITY OF HELSINKI

Tiedekunta — Fakultet — Faculty	Koulutusohjelma — Utbildningsprogram — Study programme
Matemaattis-luonnontieteellinen tiedekunta	Tietojenkäsittelytiede

Tekijä — Författare — Author

Toni Räsänen

Työn nimi — Arbetets titel — Title

Jatkuva integraatio ja mallien yhteistoiminta koneoppimissovelluksissa

Ohjaajat — Handledare — Supervisors

Prof. Jukka K. Nurminen

Työn laji — Arbetets art — Level	Aika — Datum — Month and year	Sivumäärä — Sidoantal — Number of pages
Maisterintutkielma	4.5.2020	66 sivua, 27 liitesivua

Tiivistelmä — Referat — Abstract

Jatkuvan integraation ja julkaisun (CI/CD) käytänteet ovat tulleet osaksi ohjelmistojen kehitystä osin ketterien ohjelmistokehityskäytänteiden leviämisen myötä. Jatkuva integraatio ja julkaisu pyrkivät tuomaan läpinäkyvyyttä ja seurattavuutta ohjelmistojen kehitykseen. Ne mahdollistavat ohjelmistojen kehityksen pienissä paloissa ja tukevat uusien ominaisuuksien tai korjauksien mahdollisimman vaivatonta julkaisua ja integrointia olemassa olevaan sovellukseen. Koneoppimismallien yleistyessä ohjelmistoissa ohjelmistokehityksessä tarvitaan uusia käytänteitä tukemaan mallien kehityksen tuomia uusia työvaiheita. Koneoppimismallien kehityksessä oleellisessa osassa on opetukseen käytetty data ja mallien opettamistarpeen havainnointi. Useamman mallin hyödyntäminen ohjelmistossa monimutkaistaa ohjelmiston kehitystä ja mallien toiminnan havainnointia, esimerkiksi opetustarpeen päättelyä. Jäljitettävyyden ja toistettavuuden merkitys korostuu, kun mukana on koneoppimismalleja ja niiden mukanaan tuomia uusia työvaiheita.

Tässä työssä tarkastellaan miten koneoppimismallien hyödyntäminen ohjelmistoissa vaikuttaa ohjelmistonkehityksen työnkulkuun. Pyritään selvittämään minkälaisia vaatimuksia koneoppimismallien mukaantulo asettaa jatkuvan integroinnin ja julkaisun käytänteille ja niitä tukeville CI/CD-järjestelmille. Lähemmin tarkastellaan asetelmaa, jossa kehitettävässä ohjelmistossa on kaksi koneoppimismallia. Pohditaan, miten useamman mallin keskinäinen toiminta sujuu ja minkälaisia vaikutuksia mallien vuorovaikutuksella on esimerkiksi mallien opettamiseen ja opetustarpeen päättelyyn.

ACM Computing Classification System (CCS)
Software and its engineering → Software notations and tools → Software configuration management and version control systems
Computing methodologies → Machine learning

Avainsanat — Nyckelord — Keywords

MLOps, DevOps, CI/CD, jatkuva integraatio ja toimittaminen, koneoppiminen, koneoppimismallit

Säilytyspaikka — Förvaringsställe — Where deposited

Helsingin yliopiston kirjasto

Muita tietoja — övriga uppgifter — Additional information

Ohjelmistojärjestelmien erikoistumislinja

HELSINGIN YLIOPISTO – HELSINGFORS UNIVERSITET – UNIVERSITY OF HELSINKI

Tiedekunta — Fakultet — Faculty	Koulutusohjelma — Utbildningsprogram — Study programme
Faculty of Science	Computer Science

Tekijä — Författare — Author

Toni Räsänen

Työn nimi — Arbetets titel — Title

Jatkuva integraatio ja mallien yhteistoiminta koneoppimissovelluksissa

Ohjaajat — Handledare — Supervisors

Prof. Jukka K. Nurminen

Työn laji — Arbetets art — Level	Aika — Datum — Month and year	Sivumäärä — Sidoantal — Number of pages
MSc thesis	May 4, 2020	66 pages, 27 appendice pages

Tiivistelmä — Referat — Abstract

The practices of continuous integration and publishing (CI / CD) have become part of software development with the proliferation of agile software development practices. Continuous integration and publishing strive to bring transparency and traceability to software development. They enable software development in small chunks and support the effortless release and integration of new features or fixes into an existing application. As machine learning models become more common in software, new practices in software development are needed to support the new work steps brought by the development of models. An essential part in the development of machine learning models is the data used for teaching and the observation of the need to teach the models. Utilizing multiple models in software complicates software development and observation of the operation of the models, for example, reasoning for teaching needs. The importance of traceability and repeatability is emphasized when machine learning models and the new work steps they bring are included.

This thesis examines how the utilization of machine learning models in software affects the software development workflow. The aim is to find out what requirements the introduction of models sets to continuous integration and publishing practices and the CI/CD-systems that support them. We take a closer look at a setup in which the software being developed has two machine learning models. We examine how the interaction of several models works and what such a combination of models requires in terms of the operation and development of the models. The aim is to find out what kind of effects the interaction of the models has on training the models and reasoning the need for re-training.

ACM Computing Classification System (CCS)
Software and its engineering → Software notations and tools → Software configuration management and version control systems
Computing methodologies → Machine learning

Avainsanat — Nyckelord — Keywords

MLOps, DevOps, CI/CD, jatkuva integraatio ja toimittaminen, koneoppiminen, koneoppimismallit

Säilytyspaikka — Förvaringsställe — Where deposited

Helsinki University Library

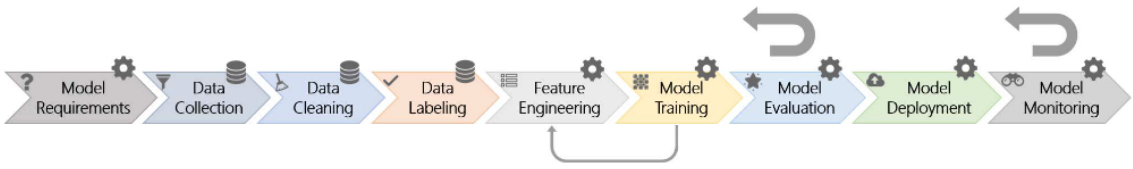
Muita tietoja — övriga uppgifter — Additional information

Software systems subprogramme

Sisältö

1 Johdanto

Ohjelmistokehityksessä kehitystyön askelten automatisointi ja toistettavuus ovat olennaisessa osassa. Automatisointia tavoitellaan jatkuvan integraation (Continuous Integration, CI) ja jatkuvan toimittamisen (Continuous Delivery tai Continuous Deployment CD) periaatteita noudattamalla. Voidaan puhua CI/CD-konfiguraatiosta, joka mahdollistaa ohjelmiston kehityksen automatisoinnin ja toistettavuuden. CI/CD-järjestelmää käyttäen yksitäisten kehittäjien lähdekoodimuutokset liitetään kehitettävään sovellukseen automaattisesti ja koodimuutosten on läpäistävä automatisoidut testit ennen muutosten liittämistä osaksi olemassa olevaa lähdekoodipohjaa. Jatkuvan toimittamisen mukaisesti nämä ohjelmistoon tehtävät muutokset pyritään viemään tuotantoon mahdollisimman nopeasti ja kestävällä tavalla (Farley ja Humble, 2010). Parhaimmillaan jatkuva integraatio ja toimittaminen nopeuttavat uusien ominaisuuksien ja ohjelmistojen saamista tuotantoon ja käyttäjien käyttöön (Forsgren et al., 2018). Jatkuvan integroinnin ja toimittamisen käytännöt ovat ajansaatossa vakiintuneet ja tulleet kiinteäksi osaksi ohjelmistokehityksen työnkulkua. Ohjelmistokehitykseen on kuitenkin alkanut tulla muutosta viime vuosina koneoppimista hyödyntävien sovellusten lisääntymisen myötä. Koneoppimismallien käyttö on lisääntynyt sovelluksissa paljolti sen ansiosta, että toiminnollisuuksia, jotka muutoin olisivat aikaa vieviä toteuttaa, on kohtuullisen vaivatonta toteuttaa koneoppimismalleja käyttäen (Amershi et al., 2019). Koneoppimiskehitystyö on luonteeltaan kokeellista, jolloin toistettavuuden ja jäljitettävyyden merkitys korostuu. Toimiva kehitysympäristö on edellytys sujuvammalle koneoppimismalleja tukevalle ohjelmistokehitykselle (Zinkevich, Martin, 2020). Tutkimusten mukaan käytännön tasolla kehittämistä kuitenkin vielä löytyy ja mitään standardimallia koneoppimismallien tukemiseen ohjelmistokehitystyössä ei vielä juuri ole (Amershi et al., 2019).



Kuva 1.1: Koneoppimismalleja sisältävän sovelluksen kehitystyön sisältämiä vaiheita.(Amershi et al., 2019)

Koneoppimismallien tuomat uudet vaiheet liittyvät pääosin mallien opetukseen. Kuvassa 1.1 nähdään kuinka koneoppimismallit tuovat lisää vaiheita ohjelmistokehitykseen. Alussa suunnitellaan malli käyttökohteen mukaan (Model Requirements). Tämän jälkeen kerätään opetusdataa mallille (Data Collection). Jotta mallit toimisivat oikein, on ne opetettava opetusdatalla, joka vastaa mahdollisimman paljon sovelluksen tulevaa käyttöympäristöä. Riippuen ympäristöstä ja datasta opetukseen käytettyä dataa on mahdollisesti siivottava ja valmisteltava ennen mallin opetusta (Data Cleaning). Mallin validointia ja testaamista varten dataa tulee myös merkata eli labeloida oikein vastauksin, jotta mallin tulosta voidaan verrata näihin vastauksiin (Data Labeling). Mallin varsinainen kehitystyö sisältää mallin toteutuksen (Feature Engineering) ja opettamisen (Model Training). Mallin validoinnissa (Model Evaluation) tarkistetaan, että malli vastaa vaatimuksia ja tarvittaessa palataan askelia taaksepäin työnkulussa. Voi olla, että tyydyttävän tuloksen saavuttamiseksi tarvitaan uutta opetusdataa. Kun malli on saatu kehitettyä tyydyttävälle tasolle, se toimitetaan tuotantoon (Model Deployment). Sen jälkeen, kun malli (osana sovellusta) on tuotannossa, sitä on monitoroitava (Model Monitoring). Mallin suoritusta on hyvä seurata, jotta voidaan reagoida ympäristön muutoksiin. Jos havaitaan, että ympäristö muuttuu eli data, jota malli käsittelee, muuttuu liian paljon siitä datasta millä malli on opetettu, malli on opetettava uudelleen. Kuvasta 1.1 nähdään, että ainakin kahdessa kohtaa työnkulkua voidaan palata takaisin uudelleen opetukseen: mallin opetusvaiheessa arvioitaessa mallin käypyyttä sekä tuotannossa, kun havaitaan monitoroinnin kautta, että malli ei pysty enää toimimaan muuttuneessa ympäristössä. Tällöin saatetaan joutua myös keräämään uutta opetusdataa ja käymään dataa valmistavat askeleet uudelleen läpi.

Tämän työn yhtenä tavoitteena on havainnoida eroja, joita on perinteisemmän sovelluksen CI/CD-systeemin ja koneoppimismalleja sisältävän sovelluksen CI/CD-systeemin välillä. Osana tätä työtä kehitettiin testisovellus, joka sisältää kaksi koneoppimismallia ja CI/CD-konfiguraation testisovellusta varten. Testisovelluksen avulla pyrittiin havainnoimaan vaatimuksia, joita koneoppimismallit asettivat sovelluskehitykselle ja erityisesti CI/CD-systeemille. Koneoppimisen myötä sovelluksen kehitystyöhön tulee lisää ulottuvuuksia, muutokset eivät enää kohdistu pelkästään sovelluksen lähdekoodiin. Koneoppimisen ytimessä on data: dataa käytetään koneoppimismallien opettamiseen. Opetetut mallit ovat syntyneet opetukseen käytetyn datan pohjalta, mallit ilmentävät opetukseen käytetyn datan kuvaamaa maailmaa (Garcia et al., 2018). Datan lisäksi koneoppimismalleja kehitetään muuttamalla niiden ominaisuuksiin vaikuttavia hyperparametrejä. Koneoppimissovellusten kehitystyöhön kuuluu lähdekoodin muutosten lisäksi mallien opettaminen, opetusdatan hankkiminen, datan siivoaminen ja versiointi sekä mallin hyperparamet-

rien säätäminen. Lisäksi opetetut mallit on hyvä versioida jollain tapaa. Testisovelluksen ja sitä tukemaan luodun CI/CD-konfiguraation esittely on luvussa 4. Toinen tämän työn tavoitteista oli tutkia kuinka koneoppimismallit vaikuttavat toisiinsa. Toteutetun testisovelluksen avulla haluttiin havainnoida, miten useampi malli toimii yhdessä. Sitä miten useamman toisistaan riippuvaisen mallin olemassaolo vaikuttaa mallien suoritukseen ja niiden kehitystyöhön ja opetukseen. Tätä varten esimerkkisovelluksessa on kaksi mallia M1 ja M2. Ne on ryhmitelty siten, että ensimmäisen mallin M1 antamaa tulosta käytetään jälkimmäisen mallin M2 syötteenä. Tällä asetelmalla pyrittiin havainnoimaan kuinka toisen mallin muutos vaikuttaa toiseen ja kuinka esimerkiksi mallien uudelleenopettamistarve voitaisiin päätellä. Tulosluvussa luvussa 5 on erilaisten testiasetelmien avulla pohdittu tätä asetelmaa tarkemmin.

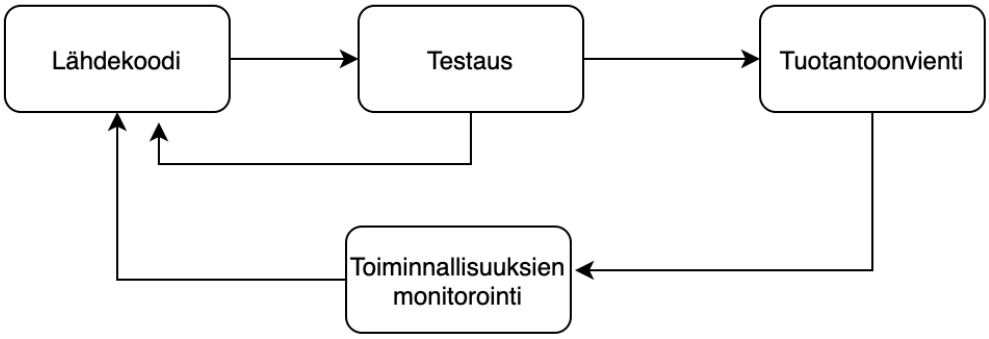
2 MLOps - DevOpsin perillinen

2.1 DevOps

DevOpsilla tarkoitetaan mallia, tapaa, jolla ohjelmistoja kehitetään. MLOps on DevOpsin kaltainen kehitysmalli, sillä lisäyksellä, että se pyrkii tukemaan koneoppimismallien kehityksen vaiheita. DevOps on läheisesti yhteydessä ketteriin ohjelmistojenkehitysmenetelmiin, joissa tavoitteena on kehittää ohjelmistoja läpinäkyvästi, suhteellisen lyhyissä kehityssykleissä (muutamia viikkoja) siten, että asiakastaho on läheisesti mukana kehitystyössä (Schwaber, Ken and Beedle, Mike, 2001). DevOpsille ei ole kovin sopivaa käännöstä suomenkielelle, voidaan puhua toimintamallista, joka noudattaa tiettyjä käytänteitä kuten jatkuvaa integraatiota (CI, Continuous Integration) ja jatkuvaa toimitusta (CD, Continuous Delivery). Käytännönläheinen määritelmä DevOpsille voisi olla, että se on joukko toimintamenetelmiä, joiden tarkoituksena on pienentää ja lyhentää ohjelmistoon tehdyn muutoksen tuotantoon saamiseen kuluvaa aikaa samalla varmistaen, että ohjelmiston laatu säilyy (Bass, Lenn and Weber, Ingo and Zhu, Liming, 2015). DevOps-menetelmien tavoitteena on nopeuttaa kehitystyön myötä syntyneen arvon siirtymistä asiakkaille, loppukäyttäjille. Tavoitteena on, että ohjelmistoon tehdyt uudet ominaisuudet tai korjaukset saadaan mahdollisimman nopeasti käyttäjille ja arvo siten realisoitua. Tämä on linjassa myös toisen ketterän ajatusmallin, Lean-menetelmän kanssa, jossa avainajatuksena on lyhyet palautesyklit ja varaston minimoiminen (Liker, Jeffrey, 2004). Ohjelmistokehityksen yhteydessä voidaan ajatella, että varasto käsittää kaikki toteutetut ominaisuudet, jotka eivät vielä ole käyttäjien käytettävissä eli tuotannossa.

Kuvassa 2.1 näkyy vaiheita, joita on DevOpsissa mukana ja joita pyritään automatisoimaan. Ohjelmiston määrittävä lähdekoodi on alkupisteenä. Lähdekoodimuutoksen jälkeen muutos viedään versionhallintajärjestelmään ja jatkuvan integraation mukaisesti seuraavassa testausvaiheessa suoritetaan automatisoidut testit. Mikäli versionhallintaan tuodut lähdekoodimuutokset läpäisevät testit, seuraa tuotantoonvienti jatkuvan toimittamisen mukaisesti. DevOps voidaan ajatella olevan myös ohjelmistoa kehittävän organisaation rajoja ylittävä menetelmä. DevOpsin erivaiheisiin voi osallistua tekijöitä organisaation eri puolilta, riippuen toki organisaation rakenteesta. Kuvassa 2.2 on ohjelmistoa kehittävä organisaation jaettu kolmeen osaan Development, QA ja Operations ja näiden kolmen

yhteistyöstä syntyy DevOps. Voidaan ajatella, että DevOps toimii organisaation rajoja ylittävänä mallina, työskentelytapana. DevOps voi koota useamman alueen ihmisiä kuten ohjelmistokehittäjiä, laadun tarkkailuun erikoistuneita henkilöitä ja ylläpitoon ja infraan erikoistuneita tekijöitä saman projektin äärelle. Tai sitten voi olla, että DevOps-hengessä organisaatiossa henkilöt tekevät työtehtäviä, jotka levittäytyvät kaikkien kolmen kuvassa näkyvän teeman ympärille. Development-osa pitää sisällään ohjelmistokehittäjiä, jotka tekevät koodimuutokset, QA-osan henkilöt vastaavat laadusta ja siten hallinnoivat testausta ja testejä. Operations-osa voi pitää sisällään tuotantoon vientiin ja tuotannon valvontaan liittyvät toimet sekä yleensä toimintoihin liittyvän infrastruktuurin hallinnoinnin. Kuvan QA-osuuden voidaan ajatella liittyvän saumattomasti jatkuvan integraation automatisoituun testaukseen. Operations-osuus voidaan ajatella liittyvän jatkuvan toimittamisen alueelle, ohjelmiston tuotantoon vientiin. DevOps-toimintatapa yhdistää ja kokoaa näitä toimijoita esimerkiksi yhden ohjelmistokehitysprojektin ajaksi.



Kuva 2.1: Ohjelmistokehityksen vaiheita, joissa DevOps mukana.

2.2 Koneoppiminen sovelluksissa

Koneoppimiselementtien (Machine Learning, ML) käyttö on yleistynyt sovelluksissa, koska se tarjoaa mainion välineen sellaisten ominaisuuksien lisäämiselle ohjelmistoihin, jotka muuten olisivat huomattavan työläitä toteuttaa (Sculley et al., 2015). Tällaisia ovat esimerkiksi erilaiset puheentunnistuselementit tai tekstinmuokkaaminen ja tulkinta, tai vaikkapa jotkin piirto-ominaisuudet. Koneoppiminen perustuu tilastomatematiikkaan, koneoppiminen perustuu menneisyydestä oppimiseen, tilastollisten tulkintojen tekemiseen menneisyyden perusteella. Tässä koneoppiminen tai koneoppimista hyödyntävä ohjelmisto eroaa

Kuva 2.2: DevOps organisaation keskiössä, kolmen osan summana nähtynä. (Wikipedia, 2020a)

luonteeltaan niin kutsutusta perinteisestä ohjelmistosta, siinä on tilastollinen komponentti mikä aiheuttaa sen, että ohjelmisto on riippuvainen datasta ja sitä kautta ympäristöstä. Koneoppimismalleja opetetaan tekemään tulkintoja käyttämällä dataa, jota on kerätty sovelluskohteesta. Oppiminen datasta tekee koneoppimiskomponenteista moniulotteisempia tavallisiin ohjelmistokomponentteihin nähden, monimutkaisuus kasvaa. Enää ei ole hallittavana vain lähdekoodi, jolla toteutetaan sovelluksen ominaisuudet, mukana on myös dataa, jota käytetään ohjelmiston opettamiseen. Voidaan ajatella, että koneoppimiselementtejä sisältävää sovellusta kehitetään lähdekoodilla ja datalla (Amershi et al., 2019).

Hallittavien asioiden määrä kasvaa, kun ohjelmistoon tulee mukaan koneoppimiskomponentteja. Koneoppimiskomponenttien mukana tulee uusia artefakteja, joita täytyy hallita ja versioida, tämä lisää monimutkaisuutta ohjelmistonkehitykseen (Staples et al., 2016). Uusia huomioitavia asioita ovat esimerkiksi koneoppimismallien opetukseen käytetty opetus- ja testidata, mallien konfigurointiin käytetyt parametrit eli hyperparametrit. Myös opetetut mallit on tallennettava ja versioitava jotenkin. (Miao et al., 2017) Koneoppiskomponentit ovat riippuvaisia ympäristöstä (Sculley et al., 2015), kun ympäristö muuttuu, opetettu malli on opetettava uudelleen. Mitä muuttuvammassa ympäristössä koneoppimismallit ovat sitä tärkeämpää on hallita ja automatisoida niiden uudelleen mukauttami-

nen eli opettaminen. Käytännössä vanha, ajastaan jälkeen jäänyt malli voi tarkoittaa sitä, että se toimii väärin. On tilanteita, joissa päivän ikäinen malli voi olla jo liian vanha ja tunnin ikäinen, tunnin välein uudelleen opetettu malli täyttää ympäristön sille asettamat suoritusvaatimukset (Hazelwood et al., 2018). DevOpsin mukanaan tuoma automaatio ja toimintamalli ei sellaisenaan enää toimi, tarvitaan lisäelementtejä, joilla hallitaan koneoppimisen mukana tulleita uusia artefakteja: dataa ja malleja. DevOpsin-käytännöillä saavutettu ennalta-arvattavuus ei toimi koneoppimiselementtien kanssa, versiointi ja ohjelmistoversioiden tuottaminen monimutkaistuu ja DevOps-käytänteet eivät sellaisenaan tue koneoppimisohjelmistojen kehitystä suoraan (Renggli et al., 2019), tarvitaan jotain uutta.

2.3 MLOps

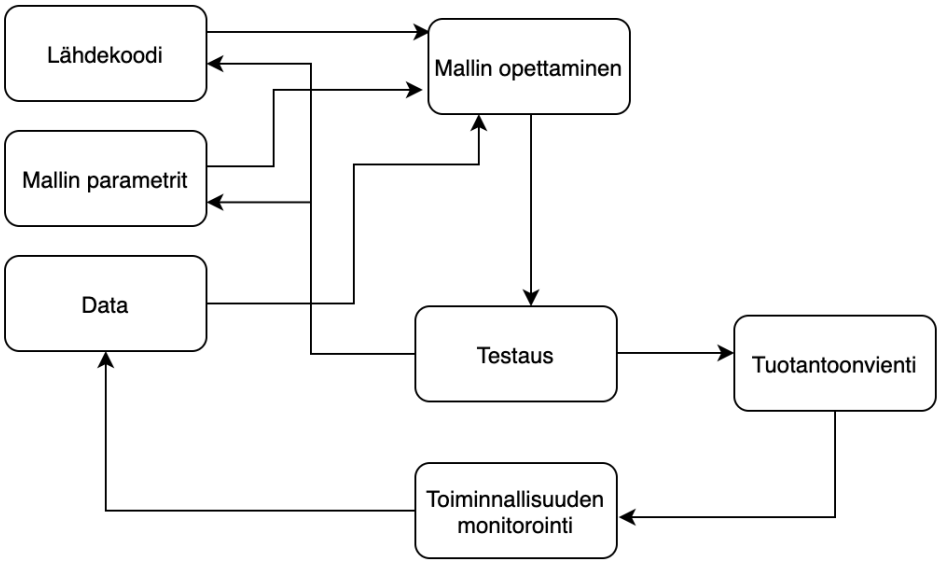
MLOps on DevOps mukautettuna tukemaan koneoppimismalleja sisältävien sovellusten kehitystyötä. Suurin ero MLOpsin ja DevOpsin välillä on mallien käsite. Koneoppimismallien mukana tulee uusia elementtejä ja työvaiheita, jotka on huomioitava (kuva 2.3). Monet haasteet koneoppimismallien kehityksessä ja siten MLOpsissa liittyvät opetuksessa ja testaamisessa käytettyyn dataan. Kuinka paljon testidataa tarvitaan, milloin tarvitaan uutta dataa ja miten data versioidaan (Renggli et al., 2019). Koneoppimismallien hyödyntäminen vaatii myös mallien tarkkailua tuotannossa, uudelleenopetustarpeen havainnointia, opetettujen mallien tallennusta ja versiointia (Rosenbaum, Sasha, 2020). Koneoppimiskomponenttien kehitystyön elinkaari poikkeaa jonkin verran perinteisen ohjelmiston kehitystyöstä. Koneoppimiselementtien kehittäminen on luonteeltaan enemmän kokeilevaa ja datavetoista. Koneoppimiselementtien kehitystyössä mallien opettaminen datalla on avainasemassa. Kehitystyö on kokeellista, koska se pitää sisällään niin paljon erilaisia muuttujia ja uuden mallin kehitystyön alussa kehittäjillä voi olla vain aavistus siitä, miten edetä, kokeiluluontoisuus korostuu ja on luontaista tällaisessa ympäristössä. (Garcia et al., 2018)

MLOpsin tavoitteena on rakentaa toiminnallinen putki (machine learning pipeline), jolla voidaan automatisoida kokonaan tai ainakin osittain mallien kehitystyössä, muutosten integroinnissa ja muutosten toimittamisessa tarvittavat askeleet. Koneoppimismallien kehitystyöhön kuuluu useita vaiheita kuten datan kerääminen ja mallin kehittäminen, mallin opettaminen, testaaminen ja validointi sekä mallin julkaisu ja jakaminen (Garcıa et al., 2020). Haasteena on mallien kehitysvaiheen kontekstin säilyttäminen, taltioiminen, siten

että jälkikäteen voitaisiin yhdistää lähdekoodi, käytetyt hyperparametrit ja mallin harjoittamiseen käytetty opetusdata. Monesti tätä tietoa ei tallenneta ja tärkeä konteksti hukataan (Garcia et al., 2018). Myös lailliset vaatimukset edellyttävät jäljitettävyyttä ja toistettavuutta, sitä miten malli on opetettu tai miten malli on päätynyt johonkin tiettyyn tulkintaan (Sridhar et al., 2018). Suoritusympäristön dokumentoiminen on myös tärkeää ja huomionarvoista, koska algoritmien ajaminen ja suoritusnopeus voivat hyvinkin vaihdella riippuen suoritusympäristön asetuksista (Sonnenburg et al., 2007). Kontekstin tallentamisen ympärille ja koneoppimiselikaaren hallinnan ja suorittamisen ympärille on alkanut tulla kaupallisia tuotteita kuten suomalainen Valohai (Valohai, 2020), tieteellisen yhteisön DEEP-Hybrid-DataCloud (Garcıa et al., 2020) tai ParallelM (Ghanta et al., 2019). Koneoppimismallien kehitystyön aikaisen kontekstin hallitseminen on tärkeää, on tärkeää pystyä yhdistämään mallien opetuksessa käytetyt eri data, hyperparametrija lähdekoodiyhdistelmät, jotta voidaan johdonmukaisesti ja toistettavasti kehittäjää ja ylläpitää malleja (Garcia et al., 2018).

Voidaan ajatella, että MLOpsilla on kaksi tavoitetta. MLOpsin tavoitteena on koneoppimistyönkulun automatisointi ja toistettavuuden, jäljitettävyyden saavuttaminen (Sugimura ja Hartl, 2018). Koneoppimistyönkulku sisältää mallien opettamisen, datan hallinnan, mallien testaamisen ja tuotantoon viennin. Toistettavuus on ollut tavoitteena koneoppimismallien parissa jo kauan ja toistettavuuden haaste on tunnistettu jo varhain (Sonnenburg et al., 2007). Toistettavuus tuo mahdollisuuden kehittää järjestelmällisesti koneoppimismalleja ja ohjelmistoa. Sonnenburg ym. nostavat toistettavuuden yhdeksi tärkeimmiksi asioiksi mallien kehittämisessä ja koko koneoppimiskentän kehityksessä. Julkaisuissa (Sonnenburg et al., 2007) ja (Hutson, 2018) toistettavuudesta puhutaan tieteellisen tutkimuksen kontekstissa, mutta saman voi ajatella pätevän myös yksittäisen ohjelmistoprojektin sisällä. Tällöin avoimuus ja toistettavuus toteutuu projektiin osallistuvien henkilöiden välillä tuoden läpinäkyvyyttä tekemiseen ja vähentää samalla riskiä siitä, että tietoa häviää tai jokin tieto olisi vain yhden ihmisen takana. Sugimura ym. (Sugimura ja Hartl, 2018) korostavat alkuperän merkitystä koneoppimiskehitystyössä. Tällä he tarkoittavat, että on tärkeää tietää, miten mallit on opetettu, mitä hyperparametrejä ja opetusdataa on käytetty. Ohjelmistot koostuvat yleensä lukuisista valmiista kolmansien osapuolten toteuttamista komponenteista, on tärkeä tietää myös näiden komponenttien versiot eli alkuperä. Toistettavuus tarkoittaa, että riippumatta ajasta voidaan olemassa olevan tiedon turvin toistaa aiemmin rakennettu asetelma. Myös saman lähdekoodin käyttö mallin opetusvaiheessa ja tuotannossa on tärkeää. Voi nimittäin olla, että malli on kehitetty ja opetettu jollain kielellä, mutta tuotantoympäristöä varten mallin toteutus kirjoitetaan uudestaan

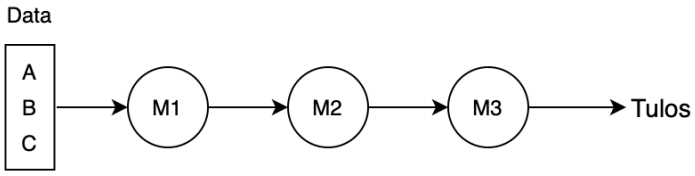
käyttäen jotain toista ohjelmointikieltä. Kielen muutos ja toteutuksen uudelleenkirjoitus altistaa virheille.



Kuva 2.3: Ohjelmistokehityksen vaiheita joissa MLOps mukana.

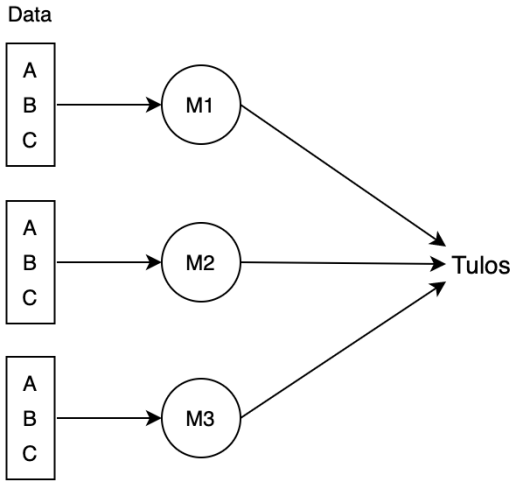
3 Koneoppimismallien yhteistoiminta

Yksi työn taustalla olleista ideoista oli havainnoida useamman mallin yhteistyötä, sitä kuinka mallin tulos vaikuttaa toisen suoritukseen. Useamman mallin yhteistyön voi ajatella olevan rinnakkaista tai peräkkäistä, ketjutettua. Tässä työssä esimerkkinä oli tapaus, jossa kaksi mallia M1 ja M2 oli ketjutettu siten että malli M2 sai syötteensä ensimmäisen mallin M1 tuloksesta (kuva 3.5). Jos mallit toimivat peräkkäin ketjussa (kuva 3.1), voidaan ajatella, että etummainen malli hienontaa saamansa syötteen, prosessoi, valmistaa syötteen seuraavalle mallille. Seuraava malli on opetettu sitä edeltävän mallin syötteellä (Rokach, 2010). Mallien yhteistyö voitaisiin järjestää myös niin, että kaksi tai useampi malli toimii rinnakkain jakaen syötteen, mallit olisivat erikoistuneet tiettyihin syötteen osiin, arvoalueisiin (Dasarathy ja Sheela, 1979). Rinnakkaisessa mallissa useampi hieman eritavalla toteutettu malli voi esimerkiksi ratkoa samaa ongelmaa (kuva 3.2) tai saman ongelman osia (kuva 3.3). Lopullista tulosta varten mallien tulokset yhdistetään. Mallit toimivat ikäänkuin ryhmässä, ne muodostavat komitean (ensemble models), jonka osatulokset lopussa yhdistetään yhdeksi vastaukseksi (Brown, 2010). Voitaisiin ajatella, että ideaalitilanne olisi sellainen, jossa tehtäisiin vain yksi malli (kuva 3.4) jolla päästäisiin hyvään tulokseen. Tämä on kuitenkin harvoin mahdollista, on todettu että yhdistämällä useampi malli saavutetaan parempia tuloksia kuin käyttämällä vain yhtä mallia (Brown, 2010).

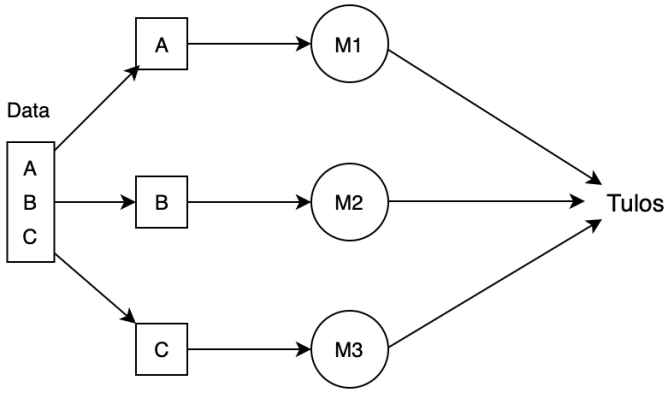


Kuva 3.1: Useampi eri tavalla toteutettu malli ketjutettuna ratkomaan ongelmaa. Ketjussa edellinen malli antaa tuloksensa syötteeksi ketjussa seuraavalle mallille.

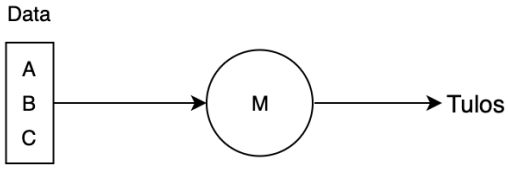
Mallien yhteistyö on ollut pinnalla jo pitkään: ajatus siitä voitaisiinko ongelmakenttä eli data jakaa useamman mallin kesken ja saavuttaa parempia ennusteita. Esimerkiksi luokitteluongelman tapauksessa on huomattu, että yksittäisen mallin keskittäminen tunnistamaan yksi luokka, ominaisuus, datasta voisi olla toimiva ratkaisu. Näitä erikoistettuja



Kuva 3.2: Useampi eri tavalla toteutettu malli ratkoo samaa ongelmaa ja mallien tulokset yhdistetään lopulliseksi tulokseksi.



Kuva 3.3: Useampi eri tavalla toteutettu malli ratkoo osaongelmia ja mallien tulokset yhdistetään lopulliseksi tulokseksi.



Kuva 3.4: Yksi malli ratkoo ongelmaa.

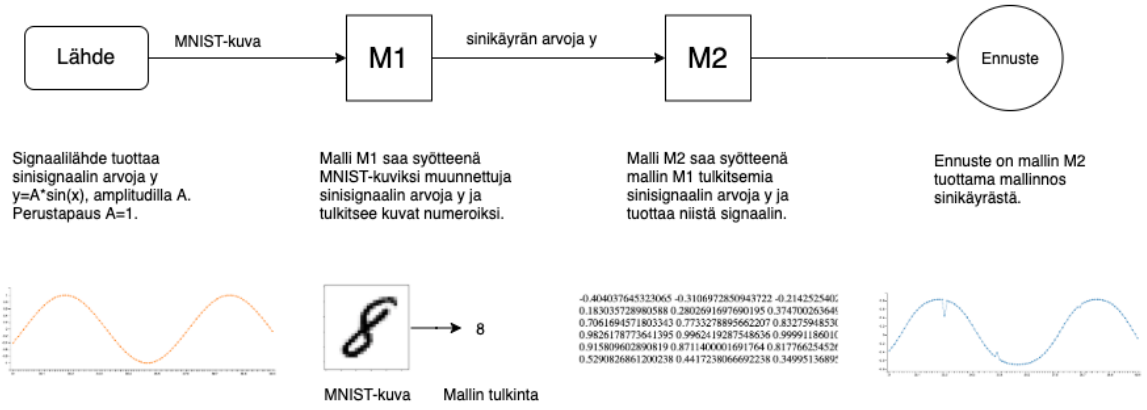
malleja ketjuttamalla saataisiin muodostettua lopullinen tulos. (Gonçalves et al., 2015) Abstraktiotasoa nostamalla päästään tilanteeseen, jossa yksittäiset mallit ovat itsenäisinä palveluina, joita voidaan yhdistää ketjuttamalla niiden tuloksia. Esimerkiksi kuvan tunnistusmallin tuottamat tunnistamistulokset voidaan antaa syötteenä jollekin tekstiä analysoivalle mallille (Algorithmia, 2019).

Ajatus, että malleista muodostetaan kokonaisuuksia jotain ongelmaa ratkomaan muistuttaa Unix-käyttöjärjestelmää ja sen perusideologiaa. Eräs Unix-käyttöjärjestelmän kantavista ajatuksista on ollut, että toiminnallisuus koostuu pienistä erikoistuneista yksiköistä, sovelluksista, jotka suorittavat suhteellisen yksinkertaisen asian mutta hyvin (Pike ja Kernighan, 1984). Koneoppimismallit voidaan nähdä samankaltaisina eriytyneinä suoritusyksikköinä, jotka ottavat syötteensä ulkomaailmasta ja tarjoavat ennusteensa ulos. Yhdistämällä malleja voidaan muodostaa kokonaisuuksia ratkaisemaan kulloinkin edessä oleva tehtävä. Tällöin nämä erikoistuneet yksiköt, mallit voidaan esimerkiksi ketjuttaa siten että ne tarjoavat syötteensä ketjussa seuraavalle ja käyttävät omana syötteenään ketjussa edeltävää mallia (kuva 3.1). Viestinvälitys ja viestin muoto sekä sisältö ovat keskeisessä roolissa toiminnallisten kokonaisuuksien rakentamisessa. Voidaan puhua viestien välityksestä jonkin ohjelmointikielen yksittäisten objektien välillä, suurempien loogisten yksiköiden kuten Unixin sovellusten tai vaikka Internetin palveluiden välillä (Kay, Alan, 1998), (Kay, 1996). Koneoppimismallit voidaan nähdä samanlaisina rakennusosina, moduuleina, joita voidaan yhdistää ratkomaan erilaisia ongelmia tilanteen mukaan. Koneoppimismallit voidaan opettaa ymmärtämään muuttuneen ympäristön muuttuneita viestejä uudelleenopetuksen kautta.

Sculley ym. mainitsevat artikkelissaan (Sculley et al., 2015), että mallien välinen riippuvuus olisi jopa kalliimpaa ja hankalampaa sovelluksen ylläpidettävyyden kannalta kuin lähdekoodin tasolla oleva riippuvuus. Vaikka erilliset mallit toimisivat halutulla tavalla niiden yhdistelmä voi toimia huonosti kokonaisuuden kannalta (Staples et al., 2016). Samoin mikäli toinen kahdesta mallista koulutetaan uudestaan uudella datalla paremmaksi voi kokonaistulos heikentyä, koska jälkimmäinen malli on opetettu toimimaan erilaisen syötteen kanssa (Garcia et al., 2018), mallien välisen viestin rakenne on muuttunut. MLOps ja koneoppimistyönkulun automatisointi voi tuoda useamman mallin yhdistämiseen konkreettisia apuvälineitä. Useamman mallin yhteistyössä mallit ovat tavalla tai toisella riippuvaisia toisensa tuloksista. Esimerkiksi ketjutetut mallit opetetaan mallia ketjussa edeltävältä mallilta saadulla syötteellä. Mallien opetustarpeen tunnistaminen, opetusdatan kerääminen ja mallien uudelleenopetus monimutkaistuvat nopeasti jo kah-

den mallin tapauksessa. Koneoppimistyönkulun automatisointi, tulosten toistettavuus ja jäljitettävyys voivat tuoda tähän monimutkaisuuteen lisää näkyvyyttä ja runkoa, jonka varassa mallien käyttö ja hyödyntäminen mahdollisesti helpottuisi.

Tämän työn esimerkkisovelluksessa (kuva 3.5) kaksi eri ongelmaa ratkomaan kehitettyä mallia on yhdistetty ketjuttamalla mallit. Molemmat mallit ratkovat oman osaongelman siten, että mallin M1 ratkaisu toimii mallin M2 syötteenä. Malli M1 ratkaisee luokitteluongelman tulkiten MNIST-kuvia (LeCun et al., 2020) numeroiksi ja malli M2 ratkaisee aikasarjaongelmaa (time series) mallilta M1 saadusta syötteestä. Mallien yhteistyön kautta esiin nousee kysymyksiä, jotka liittyvät mallien uudelleen opettamiseen ja uudelleenopettamistarpeen havainnointiin. Tämän työn tulososuudessa kappaleessa 5 tarkastellaan testiasetelman avulla näitä kysymyksiä lähemmin.



Kuva 3.5: Testiasetelman osat.

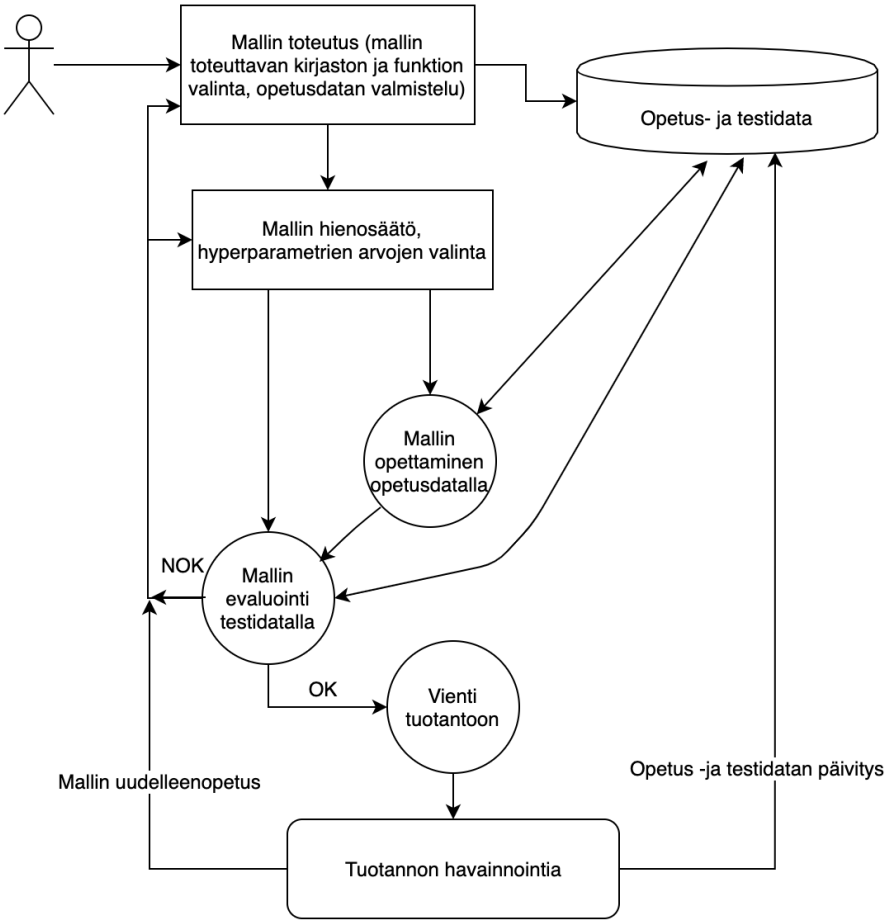
4 MLOps alustan toteutus

Tässä kuvataan tarkemmin työtä varten rakennettua CI/CD-testisysteemiä. Testisysteemi koostuu versionhallintajärjestelmästä, testisovelluksesta ja CI/CD-konfiguraatiosta. Esimerkkisovellus sisältää kaksi koneoppimismallia (kuva 3.5). Ensimmäisen mallin M1 tekemää ennustetta (prediction) käytetään toisen mallin M2 syötteenä, jonka perusteella koneoppimismalli M2 tekee ennusteensa. Tällä asetelmalla saadaan rakennettua riippuvuusketju kahden koneoppimismallin välille ja voidaan havainnoida miten toisen mallin muutos vaikuttaa toiseen. Tuloksista tarkemmin tulosluvussa luvussa 5. Esimerkkisovellus koostuu (kuva 4.2), signaalilähteestä (ulkoinen web-sovellus), koneoppimismallista M1, koneoppimismallista M2 sekä pääsovelluksesta. Pääsovellus (web-sovellus) kutsuu koneoppimismalleja ja tarjoaa http-rajapinnan ennustepyynnöille. Http-rajapinta mahdollistaa sovelluksen kutsumiseen www-selaimella tai vastaavalla http-pyyntöjen tekoon kykenevällä sovelluksella. Alla kuvataan tarkemmin kukin yllä listatuista osista. Ensin kerrotaan testisovelluksesta ja sen osista ja tämän jälkeen pureudutaan testisovellusta varten GitLab-versionhallintajärjestelmään (GitLab, 2020) tehtyyn CI/CD-konfiguraatioon. GitLab-versionhallintajärjestelmästä ja sen konfiguraatiosta tarkemmin kappaleessa 4.2 (CI/CD-konfiguraatio).

4.1 Esimerkkisovellus

Esimerkkisovelluksen avulla kokeiltiin miten koneoppimismalleja sisältävän sovelluksen kehitystyö vaikuttaa CI/CD-menetelmään. Haluttiin tutkia mitä vaiheita ja vaatimuksia koneoppimismallit tuovat kehitystyöhön ja DevOps-lähestymistapaan kehittää ohjelmistoja. Kuvassa 4.1 nähdään koneoppimismallin kehityksen vaiheita. Esimerkiksi Miao ym. (Miao et al., 2017) listaavat työnkulkuun kuuluvaksi datan keruun ja pilkkomisen, mallin luonnin (toteutus ja opetus), mallin testaamisen ja arvioinnin sekä mallin julkaiseminen. Tämän lisäksi julkaistun mallin suoritusta on syytä monitoroida jotta voidaan havaita mahdollinen uudelleenopetustarve (kuva 4.1).

Esimerkkisovellus toteutettiin Python-ohjelmointikielellä (Python Software Foundation, 2020). Merkittävä kysymys alussa oli se, kuinka tiukasti sovelluksen eri osat on integroitu toisiinsa, millä tavalla sovelluksen sisältämät koneoppimismallit on integroitu osaksi



Kuva 4.1: Vaiheita joita kuuluu koneoppimismallin kehitystyöhön.

sovellusta. Sovelluksen rakenne hyvin pitkälti määrää sen minkälainen CI/CD-systeemi sovelluksen kehitystyötä tukemaan rakennetaan. Voidaan ajatella ainakin seuraavia vaihtoehtoja sovelluksen rakenteeksi:

1. Sovelluksen lähdekoodi on kokonaisuudessaan versionhallinnassa samassa projektissa, koneoppimismallit ja pääsovellus kutsuvat toistensa metodeja.

2. Sovelluksen lähdekoodi on versionhallinnassa samassa projektissa, mutta mallit ja pääsovellus ovat eriytetty käyttäen integraatiopisteenä jotain tietovarastoa, esim. Redis tai MongoDB.

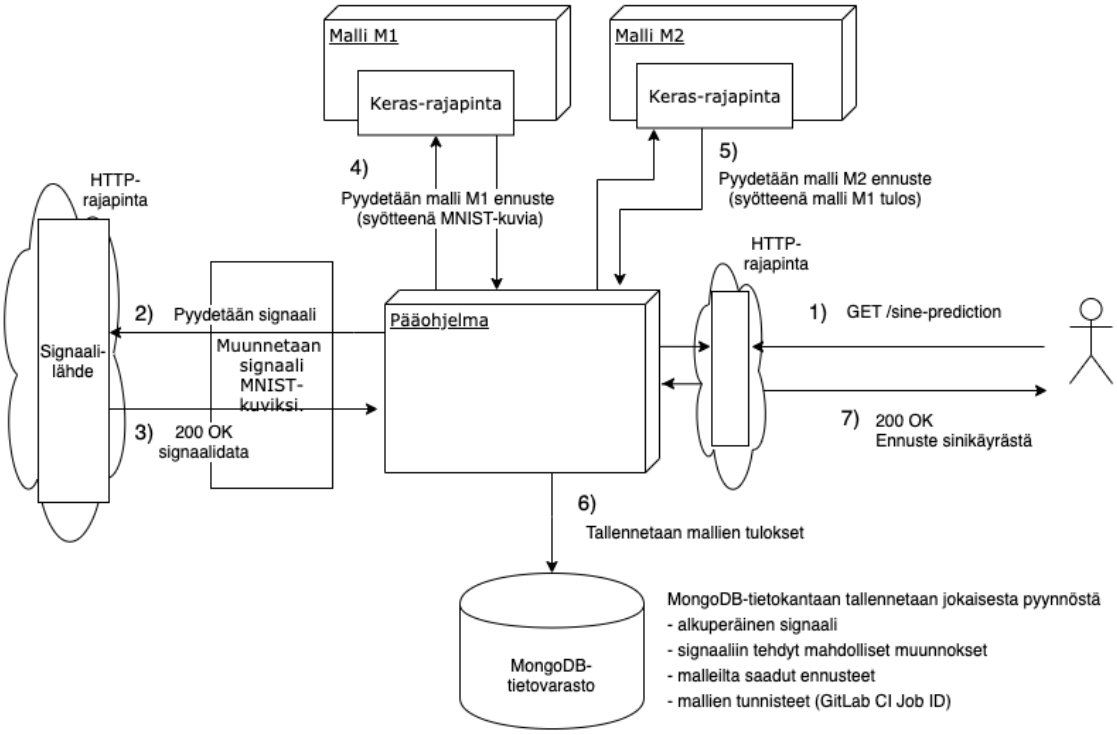
3. Sovelluksen lähdekoodi on jaettu eri projekteihin ja mahdollisesti kokonaan eri versionhallintajärjestelmään. Koneoppimismallit ja malleja käyttävä sovellus ovat omis-

> sa projekteissaan versionhallintajärjestelmässä. Integraatiopisteenä sovelluksen ja koneoppimismallin välillä on esimerkiksi http-rajapinta. Tällöin mallit tarjoavat ennusteitaan http-rajapinnan kautta, malleja käytetään http-rajapinnan yli ja ne ovat täysin erillään ja tietämättömiä niiden palveluja käyttävästä sovelluksesta. (Garcıa, 2019).

Esimerkkisovelluksessa mallit on integroitu osaksi sovellusta ensimmäisen kohdan mukaisesti: sekä koneoppimismallit, että malleja hyödyntävä sovellus ovat samassa projektissa ja siten samassa versionhallintajärjestelmässä. Sovelluksen rakenne vaikuttaa siihen minkälainen CI/CD-konfiguraatio tehdään. Voidaan ajatella sovelluksen rakenteen määräytyvän pitkälti sovellusta tekevän tahon organisaation rakenteen mukaan (Conway, 2020). Voidaan ajatella, että jos sovellusta kehittävällä organisaatiolla ei ole koneoppimisosaamista tai muutoin kyseiseltä organisaatiolta puuttuu halu koneoppimismallien kehittämiseen, mallit ostetaan muualta. Jos mallit ostetaan joltain ulkopuoliselta toimijalta, on ne hyvin todennäköisesti integroitu löyhemmin sovellukseen ja tarjoavat esimerkiksi jonkin rajapinnan kautta palveluitaan. Tässä tapauksessa mallien lähdekoodi on mallin toimittajan hallussa ja hallinnassa. Jos malleja kehitetään samassa organisaatiossa kuin niitä hyödyntävää sovellusta koko lähdekoodi saattaa olla hyvinkin tiukasti yhdessä ja mahdollisesti samassa projektissa samassa versionhallintajärjestelmässä.

Esimerkkisovelluksen rakenteeksi valittiin ajatusmalli, jossa sovelluksen koko lähdekoodi on sovellusta kehittävän organisaation omistuksessa, tällöin myös mallien opetus on osa kehitystyötä. Tässä avoimeksi jää se, kuinka lähdekoodi on organisoitu versionhallinnassa. Onko koko lähdekoodi, sovelluksen runko sekä sovelluksen käyttämät koneoppimismallit samassa versionhallinnassa samassa projektissa. CI/CD-systeemi mahdollistaa monia lähestymistapoja käsillä olevan tilanteen ratkaisuun ja tässä valittu toteutus on vain yksi mahdollisista, rakenteelliset ratkaisut ovat aina tapauskohtaisia riippuen projektista ja sovelluskohteesta. Tämän työn osalta esimerkkisovellus on kokonaisuudessaan samassa versionhallintajärjestelmässä ja samassa projektissa. Riippuen muutoksen kohteesta, siitä mihin osaan versionhallintajärjestelmässä olevaa projektia tuotu muutos kohdistuu, suoritetaan eri build- tai test-askel CI/CD-konfiguraatiossa määritellyllä tavalla. Esimerkkisovellus koostuu kolmesta loogisesta ja toiminnallisesta osasta koneoppimismallista M1, koneoppimismallista M2 sekä pääohjelmasta. Pääohjelma tarjoaa http-rajapinnan mitä kautta ennusteen voi pyytää esimerkiksi web-selaimella. Ulkoisena riippuvuutena sovelluksessa on MondoDB-dokumenttitietokanta (MongoDB, 2020) jonne tallennetaan jokaisen ennusteen tulos. Kuvassa 4.2 yleiskuva esimerkkisovelluksen komponenteista suhteineen.

Seuraavaksi esitellään kukin osa tarkemmin.



Kuva 4.2: Yleiskuva testisovelluksen komponenteista ja komponenttien väliset suhteet ja kutsujärjestys.

4.1.1 Testisovelluksen koneoppimismallit M1 ja M2

Testisovelluksessa on kaksi koneoppimismallia, joista ensimmäinen M1 on Python-kielellä ja Keras-kirjastolla (Keras, 2020) toteutettu MNIST-kuvia (LeCun et al., 2020) tulkitseva malli. Malli M1 saa syötteenä kuvia, joissa on käsin kirjoitettuja numeroita ja mallin tehtävänä on tunnistaa kuvassa oleva numero. Mallin M1 lähdekoodi on nähtävissä liitteessä C. Esimerkkisovelluksen toinen koneoppimismalli M2 on sekin Python-ohjelmointikielellä ja Keras-kirjastolla (Keras, 2020) toteutettu. Sen tehtävänä on mallintaa tulevaisuutta (Wikipedia, 2020b) saamansa syötteen perusteella. Mallin M2 saama syöte on koneoppimismallin M1 tuottamaa, mallin M1 ennusteista koottu. Mallin M2 lähdekoodi on nähtävissä liitteessä E. Tässä työssä oleellista ei ollut itse koneoppimismallit tai se, miten mallit on toteutettu. Mallien tärkeimpänä tehtävänä on auttaa hahmottamaan koneoppimismallien tuomia vaatimuksia ohjelmistokehitykseen ja mallien välisten riippuvuuksien merkitystä ohjelmistokehityksen kulkuun. Esimerkkisovelluksessa on tästä syystä

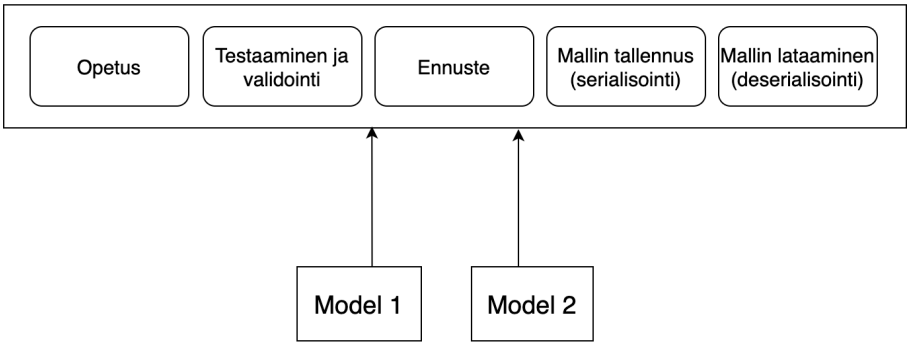
kaksi mallia, joista toinen saa syötteensä toisen ennusteesta, tällä pyritään hahmottamaan ja havainnoimaan mallien keskinäisen riippuvuuden vaikutuksia. Voidaan myös ajatella, että kahdesta mallista rakentuu yksi koostemalli. Mallit voidaan ajatella yksittäisinä tai malli voidaan ajatella kokonaisuutena, joka rakentuu useammista vahvasti toisiinsa liittyvistä malleista. Tällainen tilanne johtaa siihen, että yhden ryhmään kuuluvan mallin päivittyessä muut mallit on syytä tarkistaa ja mahdollisesti opettaa uudelleen. Ainakin on hyvä tarkistaa, että kokonaisuuden tulos säilyy oikean suuntaisena.(Sugimura ja Hartl, 2018)

4.1.2 Koneoppimismallien rajapinta

Koneoppimismalli on funktio, joka laskee tilastollisia todennäköisyyksiä ilmiöstä ja sen tuottamista arvoista, datasta. Jotta koneoppimismallin palveluita voidaan käyttää, tarvitaan jonkinlainen rajapinta mallin toteutuksen ympärille. Mallilla on jokin rajapinta, jonka kautta se on yhteydessä sovellukseen, joka hyödyntää mallin tarjoamia palveluita. Rajapinnan rakenne on tapauskohtaista ja riippuu osittain siitä, miten malli integroituu muuhun sovellukseen. Mallin elinkaareen kuuluu monia tapahtumia kuten mallin opetus, mallin testaaminen ja validointi, mallin tallennus, mallin lataaminen, ennusteen tekeminen, ennusteiden monitorointi. Ennusteiden monitoroinnissa jonkinlaisten suoritusarvojen perusteella yritetään päätellä, kuinka hyvin malli suoriutuu tehtävästään. Kaikki nämä toiminnot ovat potentiaalisia palveluita mallin rajapintaan. Testisovelluksessa molemmat mallit on integroitu kiinteästi osaksi sovellusta ja mallien osalta on päädytty toteutusratkaisuun, jossa molemmat mallit toteuttavat samanlaisen rajapinnan, jonka kautta niitä voidaan kutsua ja käyttää. Kuvassa 4.3 on kuvattu tämä mallien elinkaareen liittyviä toimintoja tukeva rajapinta.

Yleisesti ajateltuna koneoppimismallin rajapinnan osien näkyvyys tai julkisuus voi vaihdella sen mukaan millä tavalla malli on integroitu sovellukseen. Jos malli esimerkiksi tarjoaa palvelujaan http-rajapinnan yli ei tämä rajapinta tarjoa välttämättä välineitä esimerkiksi mallin uudelleenopetukseen, nämä palvelut ovat mallin sisäistä toteutusta ja ainoastaan mallin kehitysaikana käytössä. Esimerkkisovelluksen molemmat koneoppimismallit on toteutettu käyttäen avoimenlähdekoodin Keras-kirjastoa (Keras, 2020). Keras-kirjaston mallipohja tarjoaa malleille yhtenäisen rajapinnan, jossa on esimerkiksi opetukseen fit-metodi, mallin arvioinnissa käytettävien tunnuslukujen pyytämiseen evaluate-metodi, mallin tallentamiseen save-metodi ja mallin lataamiseen levyltä load-metodi. Testisovelluksessa mallien ympärille on toteutettu rajapinta, joka kutsuu taustalla Keras-kirjaston tarjoamaa

rajanpintaa.



Kuva 4.3: Mallien rajapinnan palveluita.

Testisovellukseen malleille toteutetun rajapinnan tarjoamia toimintoja:

1. Opetus. Mallin opetus käynnistetään kutsumalla komentorivillä mallia ja antamalla parametriksi opetuksen käynnistävä ”training-parametri. Opetus käynnistetään samalla komennolla sekä paikallisessa kehitysympäristössä, että CI/CD-ympäristössä GitLab-versionhallintajärjestelmässä. Testisovelluksen sinikäyrää ennustavalle mallille testauksen aloittava komento on muotoa python src/models/ML2/sine_prediction.py training ja testisovelluksen MNIST-kuvia tulkitsevalle mallile puolestaan python src/models/ML1/image_p
Mallin opetuksen aikana rakennetaan malli, luetaan opetusdata ja labelit levyltä, kutsutaan Keras-rajapinnan fit-metodia ja opetuksen jälkeen Keras-rajapinnan evaluate-metodia, joka palauttaa joitain avainlukuja muodostetusta mallista. Jos malli täyttää sille asetetut vaatimukset se tallennetaan levylle kutsumalla mallin toteuttaman Keras-rajapinnan save-metodia. Mallien laatuvaatimukset voidaan määritellä model.ini-tiedossa kohdassa [model evaluation metrics threshold values]. Kuvasta 4.4 nähdään kuinka ohjelman suoritus etenee opetuksen aikana. Kuvan vasemmassa laidassa nähdään opetuksen käynnistävä taho. Opetuksen voi käynnistää kehittäjä omalla koneellaan tai GitLab-versionhallintajärjestelmän CI/CD-konfiguraatiossa määritelty opetusaskel (GitLab Job). Opetusaskel on määritelty CI/CD-konfiguraatiossa käynnistymään siinä vaiheessa, kun versionhallintaan tuodaan muutoksia mallin toteutukseen tai mallin konfiguraation, sen hyperparametreihin. Myös uuden opetusdatan tuominen versionhallintaan aktioiva GitLab-versionhallintajärjestelmän CI/CD-konfiguraatiossa

määritellyn mallin opetusaskeleen. Mallin opetusdata on tallennettu versionhallintaan ja se luetaan levyltä tässä yhteydessä. Jatkokehitysidea olisi versioida mallien opetusdata jonnekin ulkopuoliseen järjestelmään ja lukea opetusdata mallien read_training_data_labels_from_file -metodissa versionhallintaan tallennetun viitteen kautta. Eräs mahdollinen toteutusvaihtoehto tähän on DVC-versionhallintajärjestelmä (DVC, 2020). DVC:n ideana on toimia linkkinä tai tarjota linkki git-versionhallintajärjestelmän ja isojen binääritiedostojen välille pitämällä yllä viitteitä tiedostoihin ja nämä DVC:n tarjoamat viitteet tallennetaan versionhallintaan muun ohjelmiston lähdekoodin kanssa.

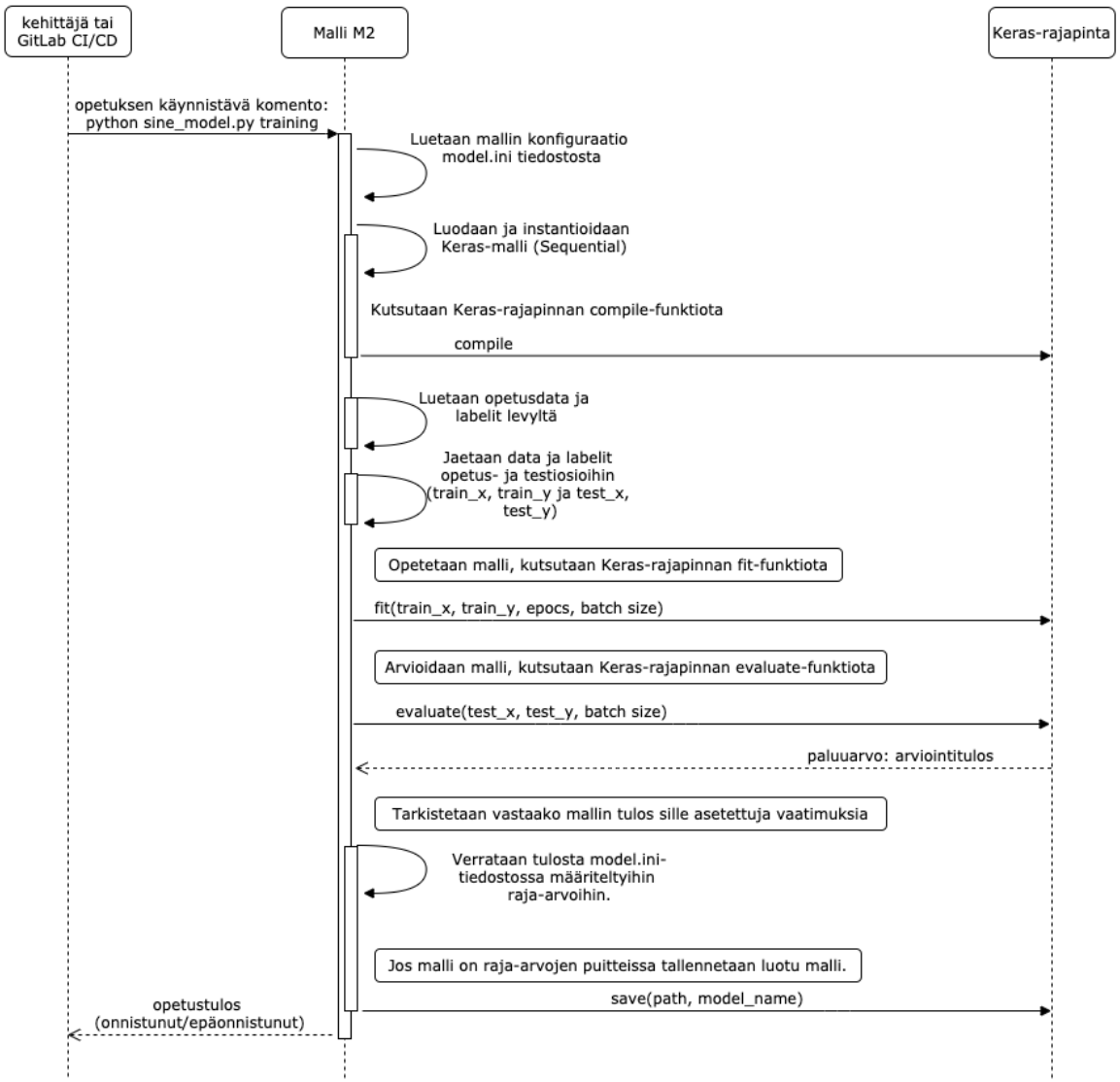
2. Testaaminen ja validointi. Mallin testaaminen ja validointi tapahtuu opetuksen yhteydessä. Malleja varten on testisovelluksessa toteutettu metodi model_satisfies_quality_constraints (a
jolle annetaan Keras-mallin evaluate-metodin palauttaman muuttujan arvo. Mallin laadulliset raja-arvot, joihin opetetun mallin tulosta verrataan voidaan antaa mallin model.ini-asetustiedossa. Kuvassa 4.4 nähdään mallin validoinnin sijoittuminen mallin opetuksen yhteydessä. Mallien validointia olisi mahdollista laajentaa, tällä hetkellä tarkistetaan vain mallin M1 yhteydessä Keras-kirjaston evaluate-metodin palauttama arvo, että arvo on parempi kuin mallin model.ini-asetustiedossa määritelty. Mallien toteutukseen model_satisfies_quality_constraints (accuracy)-metodiin voisi lisätä toiminnallisuutta, siinä voitaisiin verrata mallin suoritusta mallin aiempiin tuloksiin. Mallin aiemmat tulokset voitaisiin esimerkiksi hakea tietokannasta tässä kohdassa ja verrata niitä juuri opetetun mallin tuloksiin.

3. Ennuste. Malleilta pyydetään ennustetta kutsumalla metodia do_prediction(). Metodin alussa tarvittaessa luetaan malli levyltä ja luodaan Keras-kirjaston load_model-metodilla. Tämän jälkeen kutsutaan mallin eli Keras-kirjaston ennusteen tekevää metodia. Metodille annetaan parametrina data, jonka perusteella ennuste halutaan tehdä. MNIST-kuvia tulkitsevan mallin tapauksessa metodi on predict_classes (Kuva 4.5) ja sinikäyrää ennustavalla mallilla predict (Kuva 4.6).

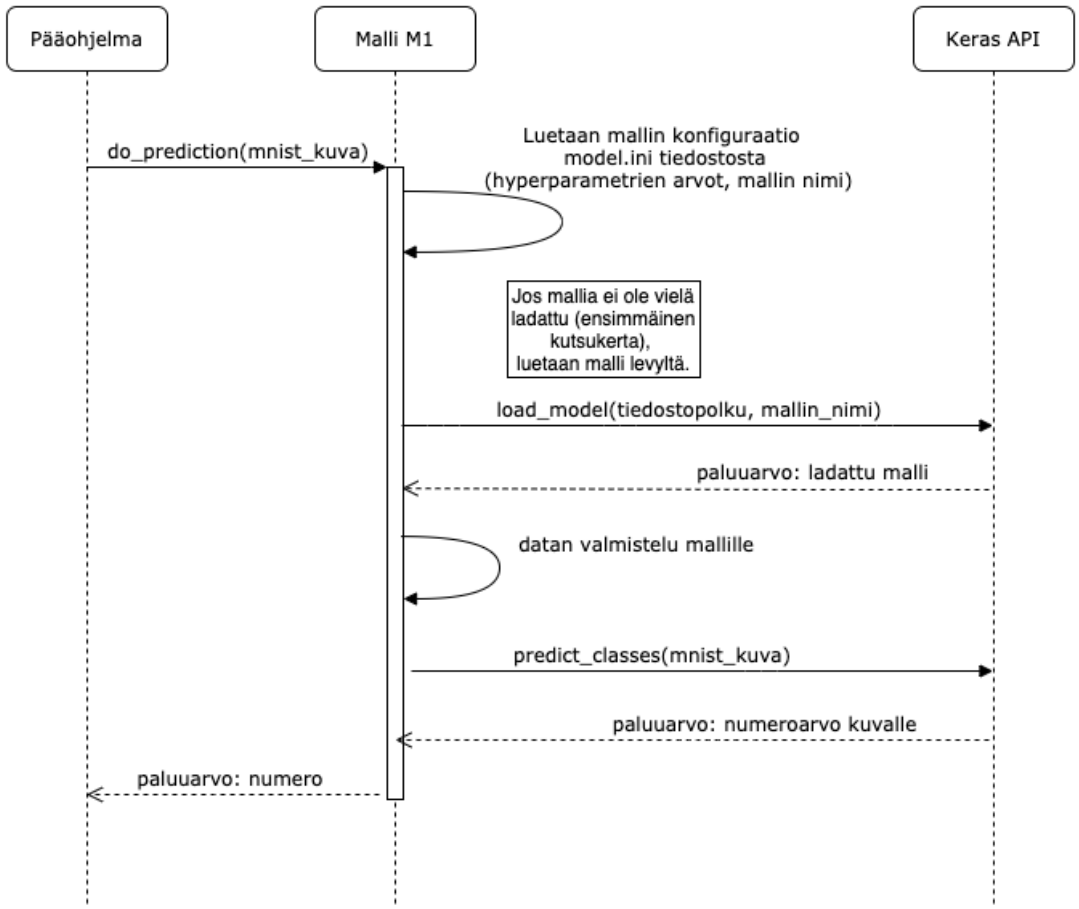
4. Mallin tallennus levylle (serialisointi). Mallin tallentaminen tapahtuu train_and_test_model -metodin lopussa, mikäli malli läpäisee opetuksen jälkeisen validaation. train_and_test_model -metodin sisällä kutsutaan Keras-kirjaston save-metodia, jossa tallennus toteutettu. Mallin nimi on määritettävissä model.ini tiedostossa. Kuvan 4.4 alalaidassa nähdään mallin tallennus levylle onnistuneen opetusjakson jälkeen.

5. Mallin lataaminen levyltä (deserialisointi). Malli ladataan automaattisesti levyltä

ennustuksen yhteydessä do_prediction-metodin alussa mikäli mallia ei ole jo aiemmin luettu levyltä muistiin. Kuvissa 4.5 ja 4.6 nähdään mallin lukeminen levyltä ennen ennusteen tekemistä. Mallit ladataan kutsumalla Keras-kirjaston tarjoamaa load_model-metodia.



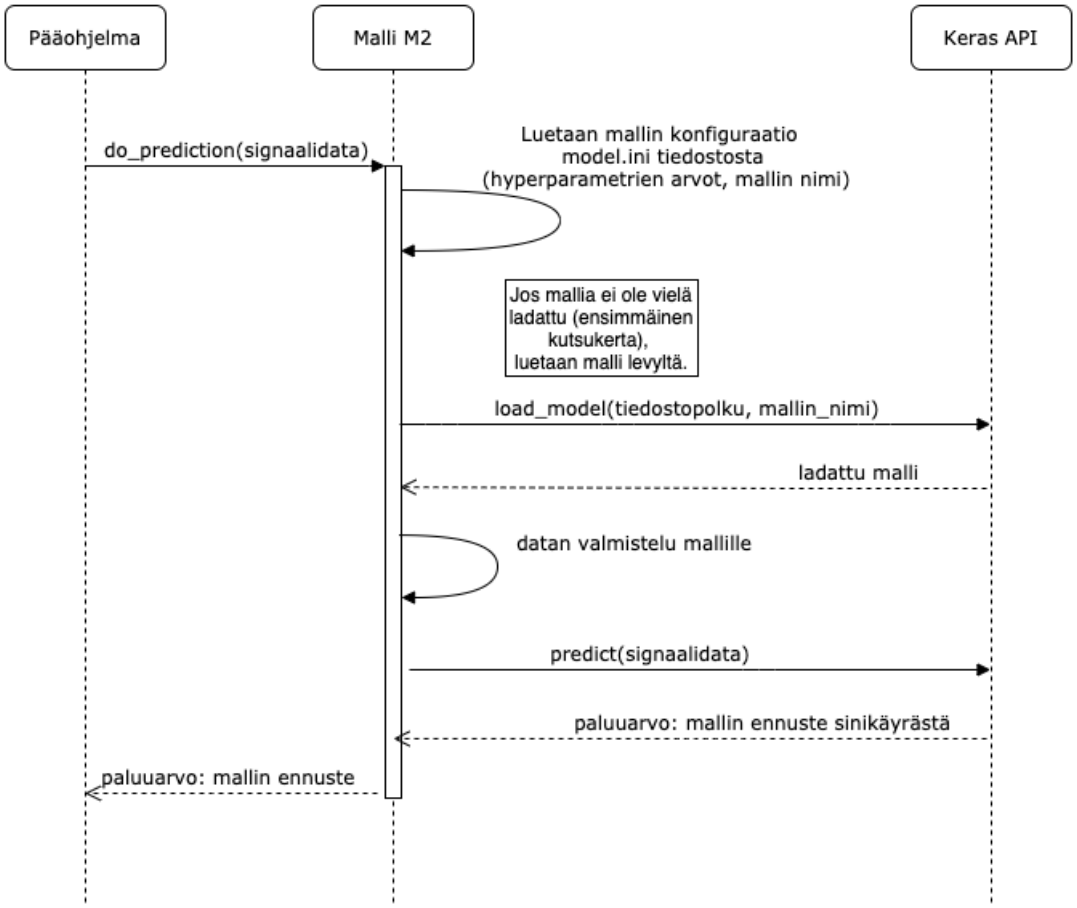
Kuva 4.4: Vaiheet mallin opetuksen yhteydessä.



Kuva 4.5: Mallin M1 toiminnallisuus ennustuksen yhteydessä.

4.1.3 Koneoppimismallien konfiguraatiotiedosto

Molemmilla malleilla on konfigurointitiedosto model.ini (Liitteet B ja D). Mallit lukevat model.ini tiedoston ja säätävät toimintaansa sen arvojen mukaan. model.ini tiedoston arvot tallennetaan versionhallintaan myös opetetun mallin binääritiedoston mukana, tämän avulla nähdään helpommin millä parametreilla malli on opetettu. Tiedostossa on kolme osiota [model info], [training config] ja [model evaluation metrics threshold values]. Model info -osiossa on yleisiä asetuksia, kuten mallin nimi ja kuvaus. Mallin kuvaukseen voi kirjoittaa esimerkiksi mallin versioon liittyviä lisätietoja. Training config -osuudessa on mahdollista asettaa arvoja mallin hyperparametreille. Tässä voidaan määritellä esimerkiksi opetuskierrosten määrä (epocs) ja batch size, eli kuinka iso pala opetusdatasta näytetään mallille kerralla. Kolmas osio konfiguraatiotiedostossa on model evaluation metrics threshold values. Tässä osassa voidaan määrittää mallille testaus- ja arviointikriteerejä, joiden



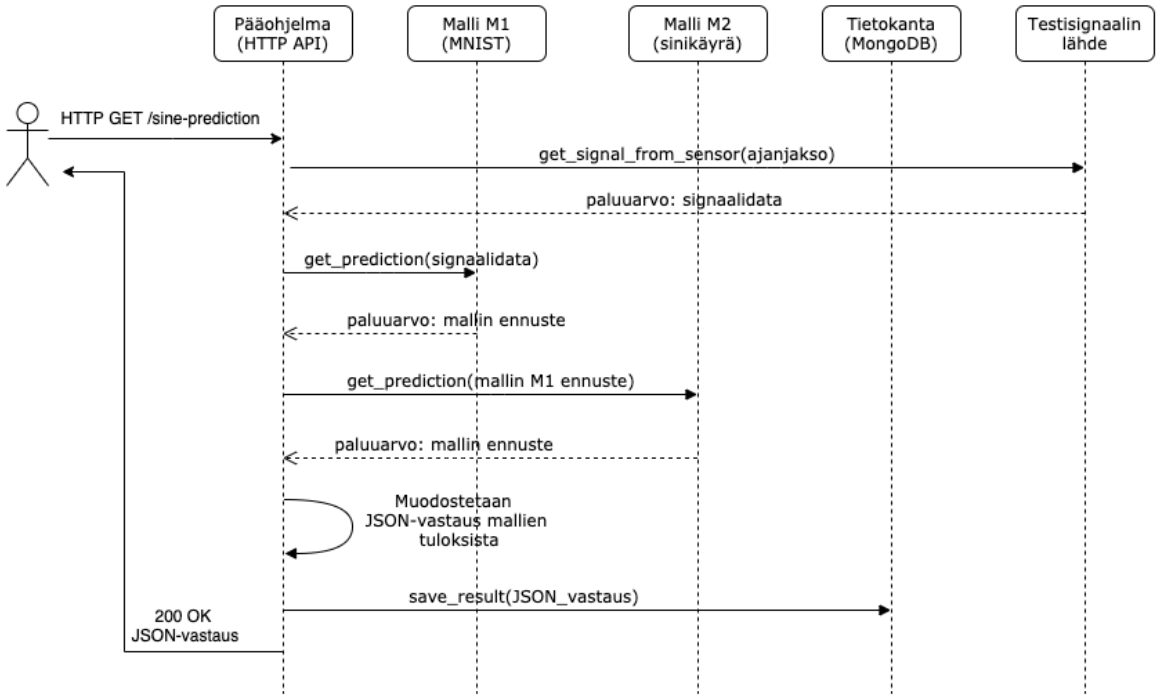
Kuva 4.6: Mallin M2 toiminnallisuus ennustuksen yhteydessä.

perusteella, tehdään päätös onko mallin opetus onnistunut. Tällä erää käytössä on vain accuracy-kriteeri ja sekin esimerkinomaisesti vain mallin M1 käytössä.

4.1.4 Pääohjelma

Pääohjelma tarjoaa http-rajapinnan ennusteiden tekoon, toisin sanoen sinikäyrän mallintamiseen. Rajapinnalle on mahdollista antaa ajanjakso, jolle sinikäyrän mallinnos tuotetaan. Ajanjakso on käytännössä x-akselin arvoja, x-akselin kuvatessa kuluvaa aikaa. Kuvassa 4.7 on havainnollistettu pääohjelman toimintaa. Pääohjelma pyytää testisignaalilähteeltä annetulle ajanjaksolle ja tämä data välitetään sovelluksen malleille pureskeltavaksi. Lopuksi tulos tallennetaan MongoDB-tietokantaan (MongoDB, 2020). Pääohjelman http-rajapinnasta on mahdollista pyytää lista kaikista tuloksista tai viimeisimmän ennus-

tepyynnön tulos. Pääohjelman lähdekoodi on nähtävissä liitteessä F.



Kuva 4.7: Sovelluksen toiminta pääohjelman näkökulmasta.

4.1.5 Ennusteiden tallennus

Sovelluksessa on MondoDB-dokumenttitietokanta (MongoDB, 2020) jonne mallien antamat tulokset tallennetaan. Tietokantaan tallennetaan kunkin signaaliennustuksen tulokset, jotta tuloksia voidaan tarkastella jälkeenpäin (kuva 4.7 alalaita). Yksittäisen ennustuskerran tulokset tallennetaan MongoDB-tietokantaan JSON-muodossa (Ecma International, 2017). Taulukossa 4.1 on listattuna mitä tietoja tietokantaan tallennetaan kustakin ennustekerrasta.

4.2 CI/CD-konfiguraatio

Seuraavassa kerrotaan CI/CD-konfiguraatiosta, joka tehtiin GitLab-versionhallintajärjestelmään testisovellusta varten. CI/CD-konfiguraatiota lähestyttiin perinteisen ohjelmistokehityksen näkökulmasta. Näkökulmaksi valittiin lähtökohta, jossa kehitettävälle ohjelmistolle,

Taulukko 4.1: Signaalivarastoon tallennettavat tiedot.

attribuutin nimi	selite
date	Ennustuspyynnön päiväys
censor_original_signal	Sensorilta saatu alkuperäinen signaali
data	Signaalidata
signal_alterations	Signaaliin testitarkoituksessa tehtyjä muunnokisia
invert_img_colors	Signaalia kuvaavissa kuvissa värit käännetty, musta pohja, valkoinen
*ml*1_*signal_prediction_result*	Ensimmäisen mallin antama ennuste
*ml*2_*sine_prediction_result*	Toisen mallin ensimmäisen mallin tuloksen pohjalta antama ennuste
*ml*1_*gitlab_job_artifact_id*	Mallin 1 tunniste jolla se voidaan jäljittää versionhallinnasta
*ml*2_*gitlab_job_artifact_id*	Mallin 2 tunniste jolla se voidaan jälittää versionhallinnasta

jossa on koneoppimismalleja mukana, rakennetaan CI/CD-systeemi muokkaamalla tai rikastamalla perinteisen ohjelmistoprojektin CI/CD-systeemiä tarvittavilta osin. Päätavoitteena oli havainnoida muutoksia, joita koneoppimismallit tuovat ohjelmistokehitykseen ja CI/CD-systeemin konfiguraatioon. Lähtökohdan valitseminen oli välttämätöntä, koska vaihtoehtojen määrä CI/CD-systeemin toteuttamiseen on laaja. Tämän työn kohdalla rajaavina tekijöinä oli aika, budjetti ja työn tavoite. Tavoitteena oli tehdä havaintoja koneoppimismallien vaikutuksista ohjelmistokehitykseen ja CI/CD-käytänteisiin.

Versionhallintajärjestelmäksi ja DevOps/MLOps-työkaluksi valikoitui GitLab-versionhallintajärjestelmä (GitLab, 2020). GitLab on suositty versionhallintajärjestelmä, joka sisältää myös CI/CD-konfiguraatiomahdollisuuden. GitLabistä on tarjolla ilmainen ja maksullinen versio, maksullinen versio sisältää enemmän ominaisuuksia ja tuotetukea. GitLabin kantavana ajatuksena on, että se yhdistää versionhallintajärjestelmän ja CI/CD-toiminnallisuuden. GitLabissa oleviin projekteihin voidaan määrittää CI/CD-askeleita ja toiminnallisuutta. Konfigurointi tapahtuu lisäämällä projektin juurihakemistoon .gitlab−ci.yml tiedosto, jossa määritellään CI/CD-toiminnallisuus. Tämä tiedosto versioituu muun projektin mukana versionhallintaan. Esimerkkisovellusta varten kirjoitettu CI/CD-konfiguraatio on nähtävissä liitteessä A. Tätä kirjoittaessa myös esimerkiksi GitHub-versionhallintajärjestelmään on tullut mahdollisuus konfiguroida CI/CD-toiminnallisuutta (GitHub, 2020) samaan tapaan kuin GitLabissä. Versionhallinnasta irrallisia CI/CD-järjestelmiä ovat esimerkiksi CircleCI (CircleCI, 2020), Travis (Travis, 2020), Jenkins (Jenkins, 2020) ja Drone (Drone, 2020). Erillinen CI/CD-järjestelmän voi joissain tapauksissa olla hyvä asia, mutta pitkälti kyse on valinnoista ja siitä mitä toiminnallisuudesta tarvitaan ja miten se halutaan to-

teuttaa. CI/CD-konfiguraation saaminen samaan versionhallintajärjestelmään, versioitumaan muun sovelluksen lähdekoodin kanssa on joskus merkittävä valintakriteeri. Riippuu myös tilanteesta, onko sovellusta kehittävässä organisaatiossa erillinen DevOps-tiimi, joka ylläpitää CI/CD-järjestelmiä. Tällöin erillinen CI/CD-sovellus voi olla hyvä vaihtoehto. Jos tilanne on se, että ohjelmiston kehittäjät ylläpitävät ja konfiguroivat CI/CD-järjestelmää voi olla selkeämpää pitää CI/CD-konfiguraatio ja lähdekoodi samassa paikassa, samassa versionhallintajärjestelmässä.

Koneoppimisohjelmistojen kehitystyönkulun automatisointiin ja hallintaan on myös maksullisia sovelluksia ja alustoja. Niitä yhdistää tavoite tehdä koneoppimistyönkulusta seurattavaa ja toistettavaa siten, että kaikesta toiminnasta jää jälki. Näin voidaan selvittää, minkälaista opetusdataa esimerkiksi on käytetty jonkin mallin kohdalla tai mitkä olivat hyperparametrien arvot opetushetkellä. Jäljitettävyys ja toistettavuus ovat tärkeitä seikkoja koneoppimistyönkulussa opetettaessa malleja (Sugimura ja Hartl, 2018). Isoilla yrityksillä on omia hieman eri tavoin rakennettuja toteutuksia saman lähtökohdan pohjalta kuten Microsoft Azure ML (Microsoft, 2020) tai Facebook FBLearner Flow (Dunn, 2020). Microsoftin palvelu koostuu monesta pienemmästä osasta ja vaikuttaa hieman epäselvältä ja on tietenkin maksullinen. Garcia ym. (Garcıa et al., 2020) ovat rakentaneet kokonaisvaltaista koneoppimisalustaa kehitystyötä tukemaan, nimenomaan tieteelliseen käyttöön. Julkaisussa mainitaan, että tarve selkeälle MLaaS-palvelulle on olemassa ja että abstraktiotaso olisi hyvä nostaa sille tasolle PaasS-tasolta (Garcıa et al., 2020). MLaas (Machine Learning as a Service) tarkoittaa palvelua, jossa voidaan helposti julkaista malleja ja ottaa toisten julkaisemia malleja käyttöön. MLaaS-palvelussa koneoppimistyönkulun askeleita on tuettu. PaaS (Platform as a Service) on matalamman abstraktiotason palvelu, jossa koneoppimiskehitystä varten hankitaan laskentaresursseja ja levytilaa ja CI/CD-toiminnallisuus on konfiguroitava itse ja yhdisteltävä eri PaaS-osat keskenään. PaaS-palvelujen hajanaisuus ja maksullisuuden taakka ja laskutuksen hajanaisuus on koettu vaikeaksi(Garcıa et al., 2020). Kokonaisvaltainen MLaaS-palvelu toisi tähän selkeyttä. Pienempiä MLaaS-toimijoita on jo olemassa esimerkiksi suomalainen Valohai (Valohai, 2020). Valohain tuotteen ydinideana on jäljitettävyys ja mallien harjoittamisen automatisointi ja rinnakkaistaminen. Jäljitettävyys toteutuu siten että kaikki toiminnot mitä suoritetaan koneoppimistyönkulun aikana, versioidaan ja tallennetaan jolloin niihin on mahdollista palata jälkeenpäin. Varsinaisesti mitään yhtä oikeaa ratkaisua koneoppimistyönkulun tukemiseen ei ole, riippuu paljon projektista ja ohjelmistosta mikä on paras ratkaisu työnkulun tukemiseen.

4.2.1 GitLabin käyttö, versionhallinta, CI ja CD

GitLab on git-versionhallintajärjestelmän (Git, 2020) päälle rakennettu web-pohjainen versionhallintajärjestelmä, joka tarjoaa myös CI/CD-toiminnallisuutta. CI/CD-toiminnallisuus konfiguroidaan projektin juureen sijoitettavassa .gitlab−ci.yml-tiedostossa. Esimerkkisovelluksen CI/CD-konfiguraation on nähtävissä liitteessä A. Kuvassa 4.8 CI/CD-toiminnallisuuden tiloja ja siirtymiä väliaskelineen. Toiminta rakentuu siten, että kehittäjä tuo jonkin muutoksen GitLab-versionhallintajärjestelmään ja riippuen muutoksen kohteesta muutos käynnistää jonkin .gitlab−ci.yml -tiedostossa määritellyn askeleen (GitLab Job). Käytännössä .gitlab−ci.yml -tiedostossa on joukko komentoja, jotka käynnistyvät tiettyjen määriteltyjen ehtojen täyttyessä. Kuvasta 4.8 voidaan nähdä, että esimerkiksi tuotaessa versionhallintaan muutos model.ini-tiedostoon käynnistyy model training -askel (GitLab Job). .gitlab−ci.yml -tiedostossa on mahdollista määritellä monipuolisesti toiminnallisuutta, tiedostossa voidaan esimerkiksi suorittaa skriptejä ja kutsua suoraan komentoja, aivan kuten oltaisiin Linux-käyttöjärjestelmän komentotulkissa. Käytännössä .gitlab−ci.yml -tiedostossa määritellyt askeleet suoritetaan GitLabin build-koneen toimesta, joka ajaa .gitlab−ci.yml -tiedostossa määritellyt askeleet docker-kontissa (Docker, 2020). Käyttäjä voi myös .gitlab−ci.yml -tiedostossa määritellä tarkalleen minkälaisessa docker-kontissa CI/CD-askeleet suoritetaan.

Rakennetun CI/CD-konfiguraation ajatus on, että testisovelluksen eri osiin tehdyt muutokset käynnistävät erilaisen suorituksen GitLab-versionhallintajärjestelmässä (kuvassa 4.8). Koneoppimismalleihin kohdistuvat muutokset käynnistävät mallin yhdistetyn opetus- ja testausaskeleen. Malli uudelleen opetetaan, mikäli mallin lähdekoodiin, mallin käyttämään testidataan tai mallin konfiguraatioon (model.ini) tulee muutoksia. Jos opetus onnistuu ja opetettu malli täyttää sille asetetut arviointikriteerit malli tallennetaan GitLabin artifaktiksi yhdessä versiotiedon kanssa sekä pilvipalvelun levylle. Mikäli opetettu malli ei täytä arviointikriteereitä opetusaskel epäonnistuu ja siitä ilmoitetaan käyttäjälle. Testisovelluksen molemmille malleille (M1 ja M2) on määritelty .gitlab−ci.yml -tiedostoon omat opetus- ja testiaskeleet. Malleista riippumattomaan osaan sovellusta tehdyt muutokset eivät käynnistä mallien opetusaskeleita vaan perinteisemmän yksikkötestiaskeleen, joka on nähtävissä kuvassa 4.8 omana suorituspolkunaan kuvan vasemmassa laidassa pääohjelmaan kohdistuvan muutoksen yhteydessä.

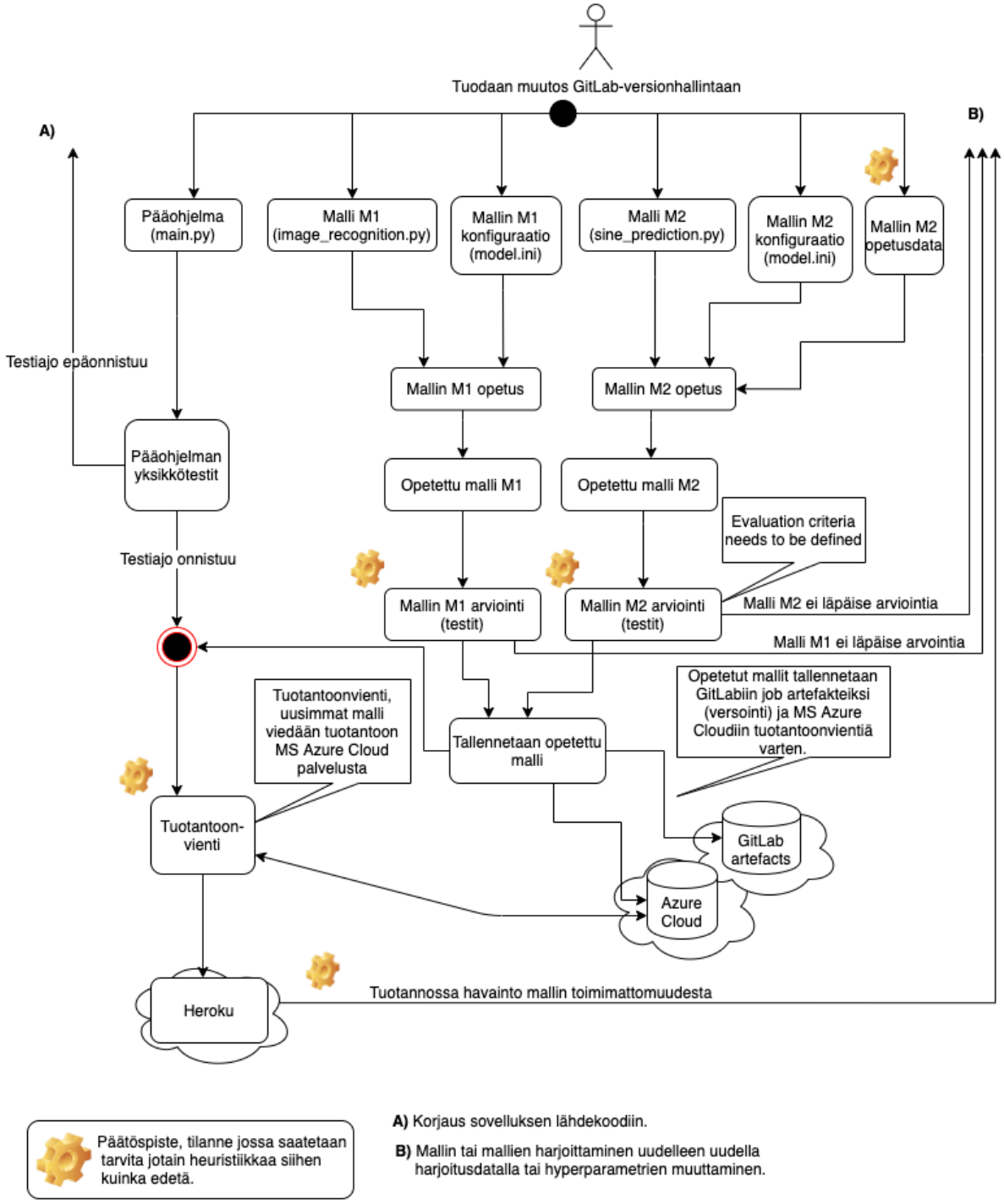
Mallin onnistuneen opetusaskeleen jälkeen opetetun mallin tila tallennetaan GitLabin artefaktiksi yhdessä opetuksessa käytetyn konfiguraation kanssa. Mallien artefaktit konfiguraatioineen tallennetaan myös ulkopuoliseen pilvilevypalveluun, josta ne voidaan viedä tuotantoon. GitLabiin tallennettavat artefaktit toimivat mallien versiointina. GitLab ei

tätä kirjoittaessa helposti tue mahdollisuutta käyttää artefakteja tuotantoon viennin yhteydessä siinä tapauksessa, että tuotantoon vienti (deploy job) ajetaan jonain muuna ajanhetkenä kuin suoraan artefaktin luoneen askeleen jälkeen. Tästä syystä käytetään ulkopuolista säilöä mallien taltiointiin. Työhön valikoitui Microsoftin Azure -levypalvelu. Onnistuneen opetusaskeleen jälkeen mallit tallennetaan verkkoon pilvipalveluun. Mallien tallentamiseen ja lukemiseen pilvipalvelusta on kaksi python-skriptiä, joita kutsutaan CI/CD-konfiguraatiosta. Skriptien lähdekoodit ovat nähtävissä liitteessä G. Yksi syy ulkopuolisen tallennuspalvelun käyttöön mallitiedostojen versioinnissa ja tallentamisessa voi myös olla mallitiedostojen koko. Riippuen projektista ja mallista mallitiedostojen koko voi olla useita satoja megatavuja, jolloin versionhallintajärjestelmän käyttäminen ei välttämättä ole toimivin ratkaisu. Tämän työn kohdalla mallien tiedostokoot ovat maltilliset, noin sadan kilotavun luokkaa. Tuotantoon vientiä varten .gitlab−ci.yml -tiedostoon on määritelty tuotantoon vientiaskel (deploy job). Tuotantoon vientiaskel on määritetty siten, että se hakee mallin tiedoston verkossa olevasta tallennuspalvelusta ja kopioi mallin tuotantoympäristöön muun sovelluksen lähdekoodin mukana. Tuotantoympäristönä käytetään tässä Heroku-palvelua (Heroku, 2020).

4.2.2 Testidatan hallinta

Esimerkkisovelluksessa testidataa säilytetään versionhallinnassa muun sovelluksen rinnalla. Todellisuudessa testidataa saattaa olla huomattavasti enemmän, riippuen toki käyttötapauksesta ja siitä millaisia mallit ovat. Sillä, että testidata on versionhallinnassa, saavutetaan se, että tiedetään millä datalla mallit on opetettu. Testidatan versiohistoria kertoo, milloin testidataa on päivitetty ja millä tavalla. Kun testidata ja mallien lähdekoodi ovat samassa versionhallintajärjestelmässä voidaan nähdä mikä lähdekoodin versio on ollut käytössä minkäkin testidatan kanssa. Ratkaisu on yksinkertainen, mutta sillä saavutetaan jäljitettävyys mallin version ja opetukseen käytetyn datan välillä (Sugimura ja Hartl, 2018). Kappaleessa 4.3 kerrotaan enemmän, kuinka testiasetelmassa käytetään testidataa ja versionhallintajärjestelmää.

Testidatan hallintaan ja versiointiin on saatavilla erilaisia ratkaisuja. Esimerkiksi DVC (Data Version Control) (DVC, 2020) tarjoaa yhden ratkaisun suuren datamäärän versiointiin. DVC sopii myös koneoppimismallien versiointiin. DVC toimii git-versionhallintajärjestelmän rinnalla. Koska git ei tue isoja tiedostokokoja DVC:n ideana on toimia eräänlaisena linkkinä ison tiedoston ja git-versionhallintajärjestelmän välissä. Isot tiedostot tallennetaan jonnekin levypalveluun ja DVC:n kautta tallennetulle tiedostolle saatu viite tallennetaan



Kuva 4.8: Yleiskuva CI/CD -toiminnallisuuden tiloista ja siirtymistä.

git-versionhallintajärjestelmään. Versionhallintajärjestelmään tallennetaan siis DVC:ltä saadut viitteet. Esimerkkisovelluksen GitLab-versionhallintajärjestelmässä kannattaisi ottaa

käyttöön esimerkiksi DVC siinä vaiheessa jos testidatan määrä eli testidatatiedoston koko kasvaisi git-versionhallintajärjestelmän kannalta liian suureksi.

4.2.3 Mallien opettaminen ja versiointi

Koeasetelman CI/CD-systeemissä mallien opetus käynnistyy kun versionhallintajärjestelmään tuodaan muutos, joka on määritelty käynnistämään opetusaskel (GitLab Job) (kuva 4.8). GitLabin .gitlab−ci.yml-tiedostoon on määritelty, että mikäli testidataan, mallin konfiguraatiotiedostoon (model.ini) tai mallin lähdekoodiin tuodaan muutos käynnistetään mallin opetusaskel (kuva 4.4). GitLab käynnistää opetusaskeleen ja suorittaa sen .gitlab−ci.yml-tiedostossa määritellyllä tavalla docker-kontissa. Askeleessa kutsutaan mallia training-parametrilla ja mallin lähdekooditiedostossa toteutettu opetusfunktio lukee levyltä opetusdatan ja suorittaa mallin opetuksen. Mikäli opetus menee onnistuneesti läpi, malli tallennetaan GitLab-versionhallintajärjestelmään yhdessä opetuksessa käytetyn konfiguraation kanssa. Tällä tavalla saadaan versioitua mallin opetustilanne. Tämän lisäksi malli tallennetaan Microsoft Azuren pilvipalveluun tuotantoon vientiaskelta varten. GitLab tarjoaa artefaktikäsitteen, jolla voidaan yhdestä build-askeleessa (job) syntynyt tuotos tallettaa versionhallintaan erilliseen artefaktisäilöön. Tämä osoittautui kuitenkin tuotantoon viennin kannalta toimimattomaksi ratkaisuksi, koska artefaktien saaminen deploy-askeleeseen oli käytännössä mahdotonta siinä tapauksessa, että tuotantoon vienti suoritetaan erillisenä toimintona irrallaan mallien opetusaskeleesta. Tavoitteena oli, että työläs opetusaskel tehtäisiin vain silloin kuin siihen on tarve ja sovelluksen tuotantoon viennissä voitaisiin käyttää ennalta opetettuja malleja silloin kuin se on mahdollista. Mallien ja opetusdatan tallentamiseen tuotantoon vientiä varten otettiin avuksi levytila Microsoft Azure-pilvipalvelusta.

Esimerkkisovelluksessa on kaksi koneoppimismallia, jotka ovat toisistaan riippuvaisia, malli M2 saa syötteensä mallilta M1. Malleja opetettaessa on otettava huomioon, miten mallin uudelleen opettaminen vaikuttaa mallin tulokseen ja sitä kautta toiseen malliin ja sen tuottamaan tulokseen. On mahdollista, että ainoaksi vaihtoehdoksi tällöin jää kehittää malleja rinnan samanaikaisesti (Amershi et al., 2019). Testisovelluksen CI/CD-systeemissä molempien mallien opetusaskeleet ovat toisistaan riippumattomia askeleita ja ne voidaan käynnistää toisistaan riippumatta. CI/CD-järjestelmä voitaisiin konfiguroida myös siten, että mallin M1 opetusaskeleen jälkeen käynnistetään automaattisesti mallin M2 opetusaskel. Tämä vaatisi hieman jatkokehitystyötä, jotta saataisiin automaattisesti tuotettua uutta opetusdataa mallille M2. Mallin M2 opetusdata on mallin M1 tuottamaa dataa, joten

dataa pitäisi saada kerättyä automaattisesti uudelleen opetetulta mallilta M1 ja tuotua se versionhallintaan mallin M2 opetusaskeleen käyttöön.

4.2.4 Mallien testaaminen ja arviointikriteerit

Testisysteemissä mallien testaaminen ja validointi on toteutettu mallien opetuksen viimeisenä vaiheena (kuva 4.4). Testisysteemissä on toteutettu kehikko mallien laadun varmistamiseen, mutta itse validointi on toteutettu vain mallin M1 osalta. Koneoppimismallin M1 kohdalla testiasetelmassa ainoa käytetty validointikriteeri on accury, eli mallin antama lukema siitä kuinka tarkasti se luokittelee saamaansa dataa. Accuracy-arvo saadaan pyydettyä Keras-kirjaston (Keras, 2020) rajapinnan kautta. Testiasetelmassa ideana on, että mallien laatukriteerit voidaan määrittää mallien konfiguraatiotiedostoissa (kappale 4.1.3). Mallin M1 konfiguraatiossa on määritelty min accuracy-arvoksi esimerkiksi 0.95 ja mallin opetusaskeleen lopuksi tarkistetaan, että malli täyttää tämän tarkkuusvaatimuksen (kuva 4.4). Mallien testaamiseen ja laadunvarmistamiseen on lukuisia mahdollisuuksia. Yksi laatukriteeri voisi olla, että uuden opetetun mallin on oltava jollain arvoilla mitattuna parempi kuin edeltäjänsä. Toisaalta tämäkään ei välttämä ole niin suoraviivaista, koska kannattaa tarkastella kokonaiskuvaa. Voi olla, että malli on osa suurempaa kokonaisuutta, kuten tässä testiasetelmassa. Tällöin ei välttämättä pystytä suoraan sanomaan onko malli parempi kokonaisuuden kannalta, vaikka se olisikin parempi kuin edeltäjänsä.

4.2.5 Mallien toimittaminen tuotantoon

Koejärjestelmän CI/CD-konfiguraatiossa on määritelty tuotantoonvientiaskel (deploy job), joka suoritetaan automaattisesti mallin M1 tai M2 opetusaskeleen suorituksen jälkeen. Tuotantoonvientaskel suoritetaan vain, jos opetusaskel on päättynyt onnistuneesti eli malli on täyttänyt asetetut laatukriteerit. On yleistä, että tuotantoonvientiaskel ei ole automaattinen vaan vaatii jonkinlaisen väliintulon ja tuotantoon viennin käynnistämisen. Tuotantoonvientiaskeleessa haetaan molempien mallien viimeisin versio verkkolevyltä pilvipalvelusta ja kopioidaan mallien tiedostot ja muu sovellus tuotantopalvelimelle Herokupalveluun (Heroku, 2020). Tämän jälkeen uusin versio sovelluksesta on käytettävissä.

4.3 Kehitystyön simulointi, testiasetelman käyttö

Seuraavassa kerrotaan, kuinka yllä esitellyllä testiasetelmalla kehitystyö sujuu, minkälaisia askeleita on otettava, kun opetetaan mallia ja halutaan siirtää uusi malli tuotantoon. Osittain tässä esitellyt asiat on käyty läpi jo tämän työn koeasetelmaa esittelevässä luvussa neljä, osittain tässä esitellään kuitenkin enemmän ajonaikaista tilannetta. Muilta osin yksityiskohtia kannattaa käydä tarvittaessa katsomassa luvusta neljä. Alla simuloidaan tilannetta, jossa tehdään muutos malliin, mallin hyperparametreihin ja opetusdataan ja opetetaan malli uudelleen. Kehitystyön vaiheista on myös muutamia kuvakaappauksia asian havainnollistamiseksi.

Koneoppimismallia kehitettäessä on mahdollista muuttaa mallin lähdekoodia, esimerkiksi muuttaa mallin käyttämää tai mallin toteuttamaa tilastollinen funktiota. Funktion toteutuksessa voidaan esimerkiksi lisätä uusia kerroksia mallin toteuttavaan neuroverkkoon. Tämän työn esimerkkisovelluksessa muutos kohdistuisi koneoppimismallin M1 toteuttavaan lähdekooditiedostoon image_recognition.py tai koneoppimismallin M2 toteutuksen sisältävään lähdekooditiedostoon sini_prediction .py. Lähdekoodit ovat nähtävissä liitteissä C ja E. Mallien lähdekooditiedostoissa viitataan mallien parametrejä sisältäviin model.ini -tiedostoihin, joissa on määritelty joitain mallien hyperparametrejä ja laatukriteerejä. Hyperparametreillä voidaan säätää esimerkiksi mallien opetuskierrosten (epocs) lukumäärää. Laatukriteerejä käytetään raja-arvoina mallien opetuksessa, esimerkkisovelluksessa on tyydytty tässä vaiheessa yhteen arvoon, joka on mallin minimitarkkuus (min accuracy). Näitä on mahdollista laajentaa kattamaan tarkempi otos laatuparametrejä, tässä yhteydessä on tyydytty esittämään tämä rakenne ilman suurempaa attribuuttien määrää. model.ini -tiedostossa on annettu myös mallista tallennettavan binääritiedoston nimi. Ajatuksena on se, että model.ini tiedostolla voidaan säätää mallia opetustilanteessa muuttamatta mallin lähdekoodia. Konfiguraatiotiedostot ovat nähtävissä liitteissä B ja D. Kolmas muutoskohta on opetuksessa käytettävä data. Opetusdata on versionhallinnassa yhdessä mallin lähdekoodin kanssa. Ajatuksena on, että kun opetustarve havaitaan versionhallintaan, tuodaan uutta dataa, tai se versio opetukseen käytetystä datasta, jolla saavutetaan haluttu opetustulos. Siinä vaiheessa, kun mallia kehittävä henkilö on saavuttanut tyydyttävän tuloksen mallin opetuksessa hän siirtää käyttämänsä datan versionhallintaan. Opetusdata koostuu kahdesta tekstitiedostosta, toisessa on sinikäyrän arvoja y ja toisessa tiedostossa on näille arvoille labelit eli kutakin y:n arvoa vastaavat x:n arvot.

Mallia kehitettäessä muutokset kohdistuvat edellä mainittuihin tiedostoihin, mallin to-

teuttavaan lähdekooditiedostoon, mallin konfiguraatiotiedostoon tai mallin opetukseen käytettyyn datan. Näiden tiedostojen vieminen versionhallintaan laukaisee versionhallinnan CI/CD-asetuksissa määritellyt askeleet ja mallin opetus ja validointi suoritetaan CI/CD-järjestelmän toimesta CI/CD-palvelimella. Onnistuneen suorituksen jälkeen malli on valmis toimitettavaksi tuotantoon. Versionhallinnassa on tämän lopputuloksena versioituna opetusdata sekä malli. Versionhallinnasta voidaan nähdä millä datalla malli on opetettu ja milloin se on tapahtunut. Tällä tavalla malli ja opetukseen käytetty data on saatu merkattua (Amershi et al., 2019) molempien kuuluesssa samaan versionhallintaan tuotuun muutosversioon (change set). Lisäksi voidaan yhdistää mallin lähdekoodi, opetukseen käytetty data ja mallin model.ini -tiedostossa määritellyt hyperparametrit sekä samaisessa tiedostossa määritellyt mallin opetuksen laaturaja-arvot toisiinsa. Versionhallinnasta nähdään lisäksi mikä malli on tuotannossa. Opetusaskeleen (GitLab job) lopputuotteena on mallin binääritiedosto ja tekstitiedosto, jossa on mallin metadata.

4.3.1 Esimerkki: opetetaan malli M2 uudella datalla

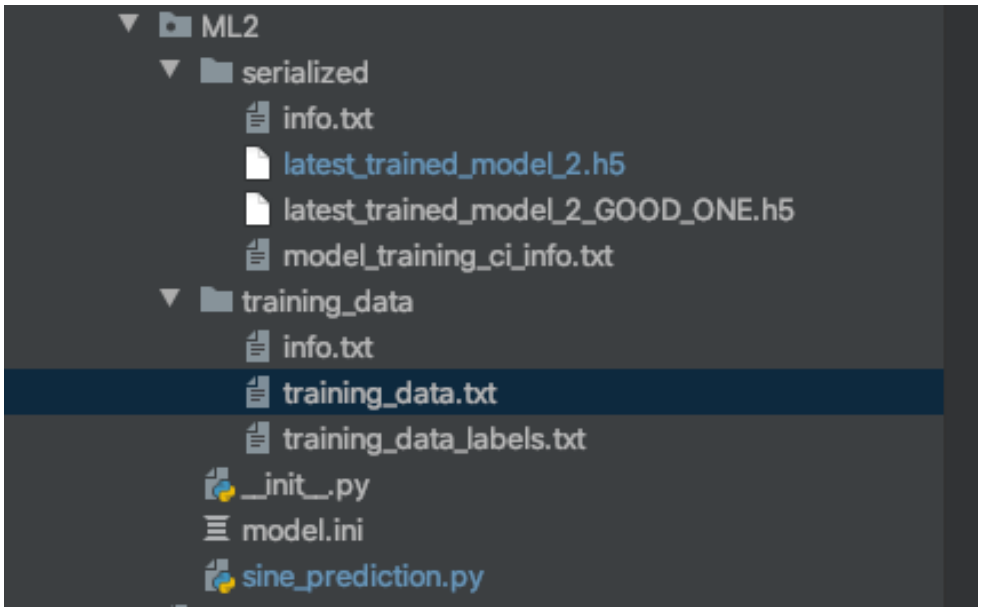
Seuraavassa esitellään testisovelluksen työnkulkua yhden esimerkkitapauksen avulla, sitä kuinka kehitystyö etenee. Esimerkkitapauksena on tilanne, jossa on havaittu tarve kohentaa koneoppimismallin M2 toimintaa. Mallia kehittävä henkilö ottaa tuotannosta uutta dataa, jota käytetään uudelleen opetuksessa. Kehittäjä testaa mallia omalla koneellaan ja saavutettuaan tyydyttävän tuloksen vie tehdyt muutokset versionhallintaan.

1. Uutta testidataa paikalliseen kehitysympäristöön.

    - Kehittäjä haalii uutta dataa esimerkiksi tuotannosta ja ottaa sen käyttöön omassa kehitysympäristössään, jotta hän voi testata mallin käyttäytymistä tällä uudella datalla.
    - Hän kopioi datan ja tarvittaessa labelit paikalliseen kehitysympäristöönsä hakemistoon, jossa data sijaitsee. Kuvassa 4.9 nähdään hakemistorakenne kehittäjän kehitysympäristössä.

2. Opetuksen käynnistäminen paikallisessa kehitysympäristössä.

    - Paikallisessa kehitysympäristössä kehittäjä käynnistää opetuksen nähdäkseen kuinka malli käyttäytyy uudella datalla. Kuvassa 4.10 nähdään kuinka mallin opetus voidaan käynnistää komentoriviltä kutsumalla mallia training-parametrilla.



Kuva 4.9: Uutta opetusdataa paikalliseen kehitykseen.

Tällöin kutsutaan mallin toteuttamaa training-funktiota, joka suorittaa opetusaskeleen käyttäen training data -hakemistossa olevaa opetusdataa ja model.ini -tiedostossa määriteltyjä parametreja. (ks. opetuksen askeleita kuvasta 4.4)

- Kehittäjä voi asettaa mallille kriteerejä, jotka hän toivoo täyttyvän. Nämä määritetään model.ini -tiedostossa. Malli toteuttaa opetusmetodissaan toiminnallisuuden, jossa mallia verrataan opetuksen valmistuttua model.ini -tiedostossa annettuihin vaatimuksiin ja jotka malli läpäisee tai ei. Tulos tulostetaan kehittäjälle opetuksen päätyttyä ja kehittäjä voi tuloksesta riippuen opettaa mallia edelleen tai tyytyä tulokseen ja harkita muutosten vientiä versionhallintaan. Kuvassa 4.11 nähdään opetusaskeleen päättymisen jälkeen kehitysympäristössä tulostettava tulos.

3. Parametrien säätäminen model.ini -tiedoston avulla

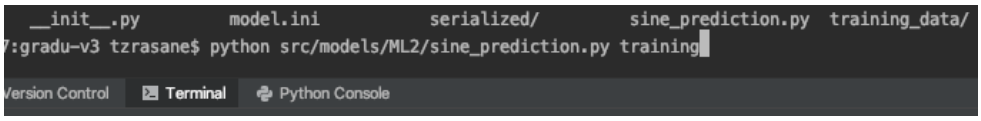
   - Kuvassa 4.12 on esitelty kullekin mallille määritelty model.ini-tiedosto, jossa on parametreja joilla mallia voidaan konfiguroida. Tarkoituksena on, että tässä tiedossa on lähdekoodista irrallaan arvoja, joita kehittäjä voi muuttaa kehittäessään sovellusta ja mallia. Nämä tiedot versioituvat myös mallin mukana ja ne otetaan mukaan mallista tallennettavan artefaktin rinnalle.
   - Kehittäjä säätää säädettävissä olevia parametreja paremman tuloksen toivossa ja ajaa edellä esitellyn opetusaskeleen muutosten jälkeen.

4. Muutosten versiointi js vienti versionhallintaan

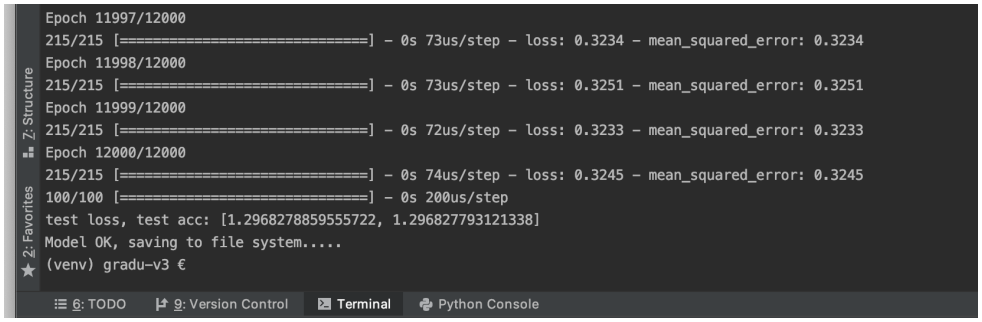
    - Kun kehittäjä on saavuttanut paikallisessa kehitysympäristössään haluamansa tuloksen hän vie muutokset versionhallintaan. Toisin sanoen vie haluamansa muutokset git-versionhallintajärjestelmään ja edelleen GitLab-versionhallinnassa olevaan projektiin.
    - Malliin tehdyt muutokset käynnistävät GitLab-versionhallintajärjestelmässä mallin opetusaskeleen (GitLab Job). Kuvassa 4.13 nähdään muutoksen viennin käynnistämä koneoppimismallin M2 opetusaskel GitLabissä-versionhallintajärjestelmässä. Tässä yhteydessä suoritetaan käytännössä samat askeleet, jotka kehittäjä on ajanut omassa paikallisessa kehitysympäristössään. Mikäli opetus ja validointi menevät onnistuneesti läpi mallista syntyy GitLab artefakti ja malli tallennetaan verkkolevylle Microsoft Azure pilvipalveluun. Opetusaskeleen (GitLab Job) lokista kehittäjä näkee, että opetusaskel on mennyt onnistuneesti läpi (kuva 4.14).

5. Uuden mallin vieminen tuotantoon

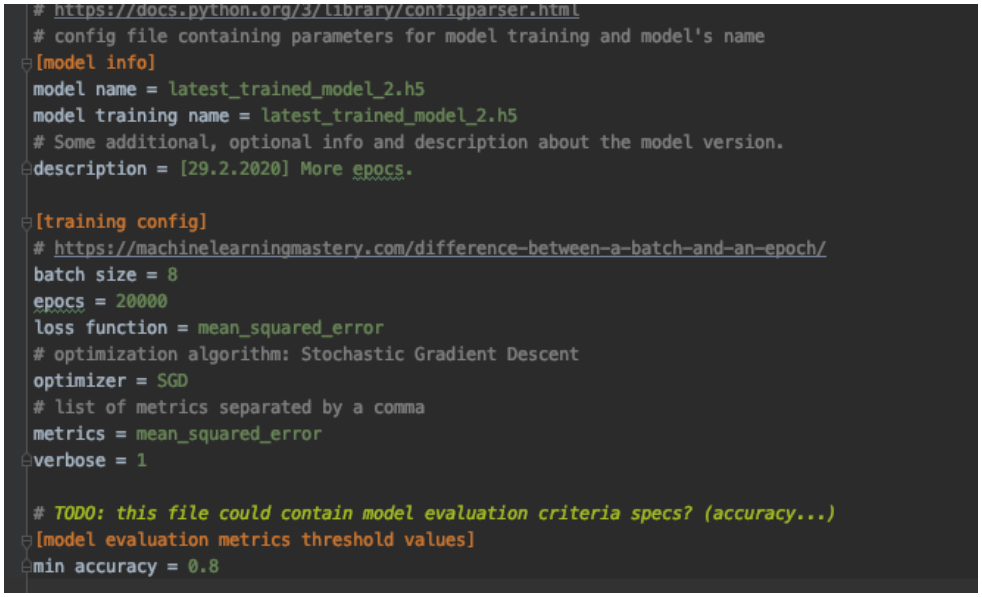
    - GitLab-versionhallintajärjestelmän CI/CD-konfiguraatiossa on mahdollista määritellä, tehdäänkö tuotantoon vienti automaattisesti muiden askelten jälkeen. On mahdollista ja myös yleinen käytäntö, että tuotantoon vientiä ei tehdä automaattisesti vaan harkinnan mukaan manuaalisesti. Tuotantoonvientiaskeleen (deploy job) lokista kehittäjä näkee onko suoritus onnistunut (kuva 4.15). Onnistuneen tuotantoonvientiaskeleen jälkeen uusi, opetettu malli ja sitä käyttävä sovellus on tuotannossa Heroku-pavelussa.



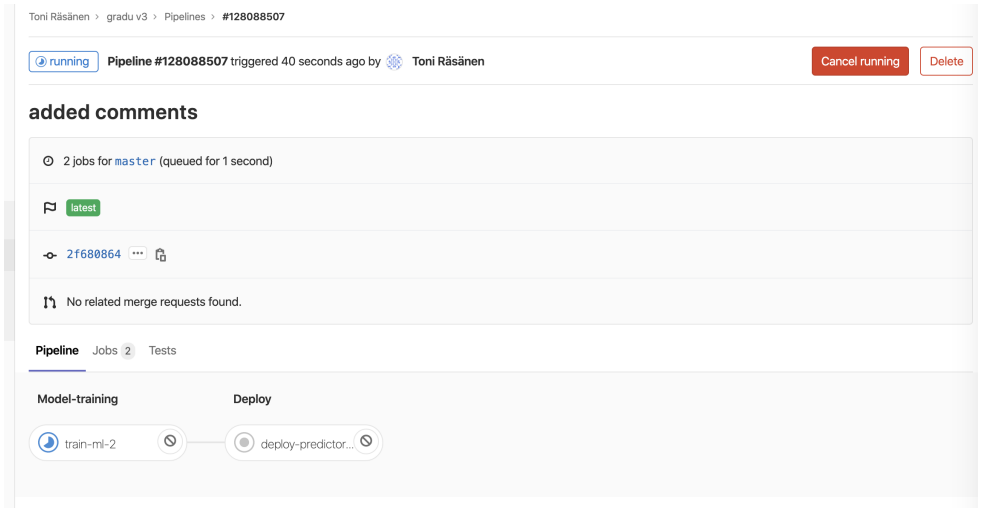
Kuva 4.10: Mallin opetuksen käynnistäminen paikallisessa kehitysympäristössä komentoriviltä.



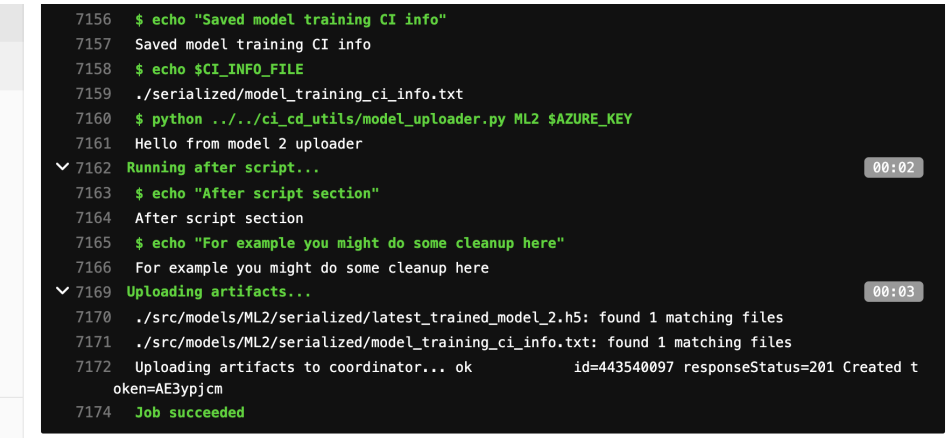
Kuva 4.11: Mallin opetustulos opetuksen jälkeen paikallisessa kehitysympäristössä.



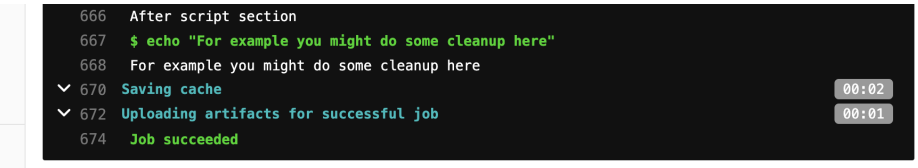
Kuva 4.12: Mallin parametreja model.ini tiedostossa paikallisessa kehitysympäristössä.



Kuva 4.13: Näkymä GitLab versionhallinnassa paikallisen koodimuutoksen toimituksen jälkeen. Mallin M2 opetusaskel on käynnistynyt.



Kuva 4.14: GitLabin opetusasekeleen lokia.



Kuva 4.15: GitLab versionhallintajärjestelmän tuotantoonvientiaskeleen loki.

5 Havaintoja kahden mallin yhteistoiminnasta

Tässä luvussa esitellään tuloksia, joita saatiin testiasetelman pohjalta. Testiasetelma koostui GitLab-versionhallintajärjestelmästä ja testisovelluksesta. GitLabiin oli konfiguroitu CI/CD-toiminnallisuus testisovellusta varten. Testisovelluksessa oli kaksi koneoppimismallia, jotka toimivat yhteistyössä. Koneoppimismalli M1 sai syötteenään MNIST-kirjaston (LeCun et al., 2020) kuvia ja muunsi niitä numeroiksi, mallin tehtävänä oli luokitella kuvat oikein numeroiksi. Toinen koneoppimismalli M2 oli aikasarjaa (timeseries) ennustava malli, se pyrki mallintamaan sinikäyrää saamastaan syötteestä. Malli M2 sai syötteensä koneoppimismallilta M1. Testiasetelmaa kokeiltiin viidellä erilaisella koetapauksella, jotka on listattu taulukossa 5.1. Koetapauksissa muutettiin koeasetelman komponenttien toimintaa, jotta saatiin luotua erilaisia asetelmia. Asetelmilla haluttiin kokeilla ja havainnoida miten mallit toimivat suhteessa toisiinsa, kun vaikkapa signaalia muutetaan, jolloin data ei enää vastaa opetuksessa käytettyä dataa. Pyrittiin myös havaitsemaan miten mallit vaikuttavat toisiinsa, kuinka ne ovat riippuvaisia toisistaan ja voidaanko toisen mallin tuloksesta päätellä jotain, esimerkiksi opetustarve.

Datan muuttuminen (data drift, concept drift) on tunnettu ongelma koneoppimisessa (Sethi ja Kantardzic, 2017). Datan muuttuessa malli ei pysty enää samalla tavalla antamaan ennustetta, mallin kyky tunnistaa dataa, datan ominaisuuksia heikkenee (Pinto et al., 2019). Voidaan ajatella, että data voi muuttua kahdella tapaa mallin näkökulmasta. Mitattava ilmiö, josta dataa saadaan, ilmiö, jota mitataan voi muuttua ajan kuluessa jolloin datan jakauma ei enää ole sama kuin mallin opetuksessa käytetyssä datassa. Toinen vaihtoehto voi olla mittausvirhe, dataa tuottavan lähteen mittauskyky heikkenee jollain tapaa riippumatta mitattavasta ilmiöistä ja tällöinkin mallin näkökulmasta data muuttuu. Käytännössä kyseessä voi olla esimerkiksi ilmiön havainnointiin käytettävän mittalaitteen kuluminen tai likaantuminen ajan saatossa. Tämä näyttäytyy muuttuneena datana, jolloin seuraus on samanlainen kuin tilanteessa jossa mitattava ilmiö muuttuu, malli on kykenemätön tekemään oikeita johtopäätöksiä ja havaintoja saamastaan datasta (Liu et al., 2017).

Haasteena on tunnistaa, kumpi vaihtoehdoista on kyseessä, kun havaitaan mallin ennus-

Taulukko 5.1: Testiasetelmat ja niissä tehdyt variaatiot.

Testitapaus	Signaali	Malli M1	Malli M2
Testitapaus 1	Ei muutosta	Hyvä	Hyvä
Testitapaus 2	Ei muutosta	Aluksi heikko, opetetaan uudelleen paremmaksi.	Alussa opetettu heikon mallin M1 datalla ja uudestaan paremman mallin M2 datalla
Testitapaus 3	Muutos (amplitudi kaksinkertainen)	Hyvä, opetettu alkuperäisellä signaalilla	Hyvä, opetettu alkuperäisellä signaalilla
Testitapaus 4	Muutos (värien kääntö MNIST-kuvissa)	Hyvä, opetettu alkuperäisellä signaalilla	Hyvä, opetettu alkuperäisellä signaalilla
Testitapaus 5	Muutos (amplitudi kaksinkertainen ja värien kääntö MNIST-kuvissa)	Hyvä, opetettu alkuperäisellä signaalilla	Hyvä, opetettu alkuperäisellä signaalilla

tuskyvyn heikkeneminen. Onko mitattava ilmiö muuttunut vai onko mittauksessa mahdollisesti jotain vikaan. Jälkeenpäin voidaan yrittää päätellä, kummasta on kyse, kun tiedetään havainnoidun ilmiön toteutuma. Tällöin käytössä voi olla merkattua (labeled) dataa tai käydään tarkastamassa mittalaitteet ja todetaan niiden kunto. Esimerkiksi (Pinto et al., 2019) ja (Ghanta et al., 2019) ja (Brzezinski ja Stefanowski, 2013) ja (Sethi ja Kantardzic, 2017) ovat tutkineet ja esittäneet ideoita, kuinka reaaliaikaisesti voitaisiin havaita datan muuttuminen ja mallin uudelleenopetustarve, kun kyseessä on data, josta ei ole saatavilla oikeaa vastausta eli merkattua versiota heti. Datan merkkaaminen on työläs ja aikaa vievä vaihe ja tehtävän vaativuus riippuu paljolti siitä, minkälaisesta datasta on kyse. Merkatun datan saaminen voi kestää viikkojakin (Pinto et al., 2019). Jos datan muutos johtuu mitatun ilmiön muutoksesta, vaaditaan mallin opettamista uudelleen, muuttuneen ilmiön mukaiseksi käyttäen uutta opetusdataa. Opetustarpeen tai laitteiston korjaamistarpeen havaitseminen riippuu siitä, kuinka usein ja millä tavalla mallin ennustetta pystytään vertaamaan todellisuuteen, toteumaan. Opetustarpeen havaitsemisen kannalta parasta olisi reaaliaikainen havainnointi, mutta tämä riippuu toki myös siitä, minkälaisessa ympäristössä malli on. Reaaliaikainen mallin havainnointi voi olla mahdotonta tai liian kallista. Käytännön tasolla mallien monitoroinnin voi toteuttaa esimerkiksi kirjoittamalla mallin tulokset lokiin ja vertaamalla niitä toteuman kanssa (Li et al., 2017).

5.1 Suoritusympäristöstä

Tässä esitellään ympäristö, jolla alla esitellyt tulokset on saatu. Aluksi kloonataan versionhallinnasta testisovelluksen projekti. Ainoa vaatimus ajoympäristölle on, että Docker (Docker, 2020) on asennettuna, tämä koskee Mac ja Windows koneita, joihin Docker-tuki on asennettava. Seuraavassa listattu tarvittavat komennot ja tiedostot tulosten muodostamiseen. Testisovelluksen käynnistävät komennot tarkoitettu suoritettavaksi testisovellusprojektin juuressa komentoriviltä.

1. Testisovelluksen käynnistys (web-sovellus ja MongoDB-tietokanta)
    - docker−compose build
    - docker−compose up
2. Testisovelluksen hakemistoja ja tiedostoja
    - src/models/ML1/model.ini
    - src/models/ML1/image_recognition.py
    - src/models/ML2/model.ini
    - src/models/ML2/sine_prediction.py
    - src/models/ML2/training_data/training_data.txt
    - src/models/ML2/training_data/training_data_labels.txt
3. Testisovelluksen koneoppimismallien opetus
    - python3 −m venv venv; source venv/bin/activate; pip install −r requirements.txt
    - python src/models/ML1/image_recognition.py training
    - python src/models/ML2/sine_prediction.py training
4. Testisovelluksen osoitteet, ennusteen pyytäminen ja tuloslistaus
    - http://localhost:8000/sine−prediction/from/31/to/41/freq/1/amp/1/inv/no
    - http://localhost:8000/sine−prediction/front/latest.html
    - http://localhost:8000/sine−prediction/front/results.html

5.1.1 Mallin M1 opettaminen

Mallin M1 opettaminen tapahtuu muuttamalla mallin hyperparametrien batch size ja epocs arvoja. Nämä ovat mallin konfiguraatiotiedostossa model.ini [training config] -osuuden alla. Hyperparametrien muuttamisen jälkeen opetus käynnistetään komennolla python src/models/ML1/image_recognition.py training. Mallin M1 opetusdata on muuttumaton se käyttää opetusdatanaan MNIST-kirjaston (LeCun et al., 2020) tarjoamaa dataa ja hakee sen opetuksen yhteydessä verkosta Keras-kirjaston (Keras, 2020) rajapinnan kautta.

5.1.2 Mallin M2 opettaminen

Mallin M2 opettaminen tapahtuu samalla tapaa kuin mallin M1 eli muuttamalla mallin hyperparametrien batch size ja epocs arvoja. Nämä ovat mallin konfiguraatiotiedostossa model.ini [training config] -osuuden alla. Tämän lisäksi mallin M2 opetusdataa voi muuttaa. Malli M2 käyttää opetusdatanaan mallin M1 tuottamaa syötettä. Mallin M2 opetusdata otetaan niin sanotusti tuotannosta mallin M1 tuloksesta, opetukseen käytetään tuotantodataa. Yllä kappalessa 5.1 on listattu testisovelluksen oleellisia tiedostoja ja muunmuassa mallin M2 opetusdatan sijainti. Mallin M2 opetusdata asetetaan kopioimalla mallin M1 antama tulos halutulta ajanjaksolta src/models/ML2/training_data/training_data.txt tiedostoon.

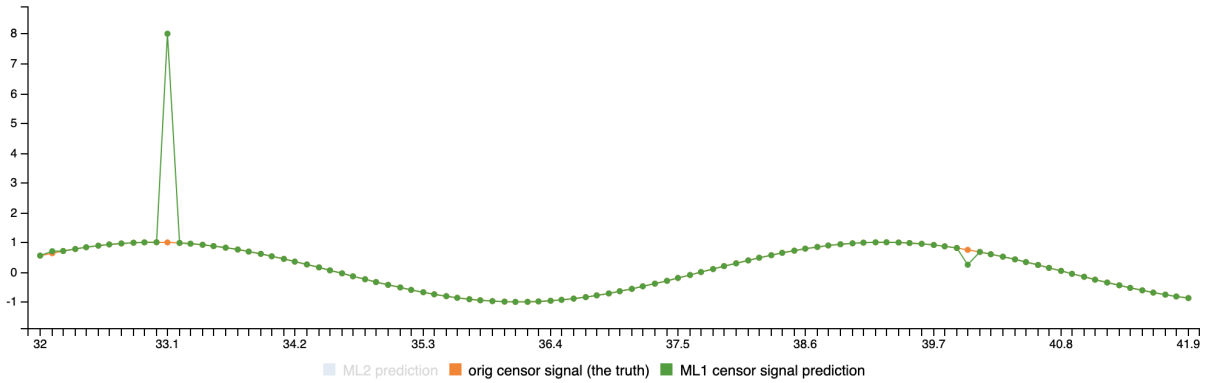
1. Pyydetään ennuste ajanjaksolle 0-31
    - http://localhost:8000/sine−prediction/from/0/to/32/freq/1/amp/1/inv/no
    - saadaan tulos ja kopioidaan arvot mallin M1 tuloksesta (ML1 RESULT: . . .) selaimen sivulta mallin M2 training_data . txt-tiedostoon mallin M2 alle training_data-hakemistoon. Näin meillä on tuore opetusdata mallille M2 otettuna mallin M1 tuloksesta.
    - Tarkastetaan, että labelit vastaavat dataa training_data_labels . txt. Labelit ovat x:n arvoja ja data on y:n arvoja. Tarkastetaan että molempia on sama määrä ja että ne ovat ”samalta ajalta”.
2. Ajetaan mallin M2 opetusaskel projektin juuressa
    - python /src/models/M2/sine_prediction.py training

3. Tarkistetaan mallin M2 opetustulos kuvaajasta
    - http://localhost:8000/sine−prediction/front/latest.html

Mallien opetuksen jälkeen käynnistetään testisovellus uudelleen ajamalla komennot docker−compose build ja docker−compose up, jotta uusdet mallit saadaan sovelluksen käyttöön.

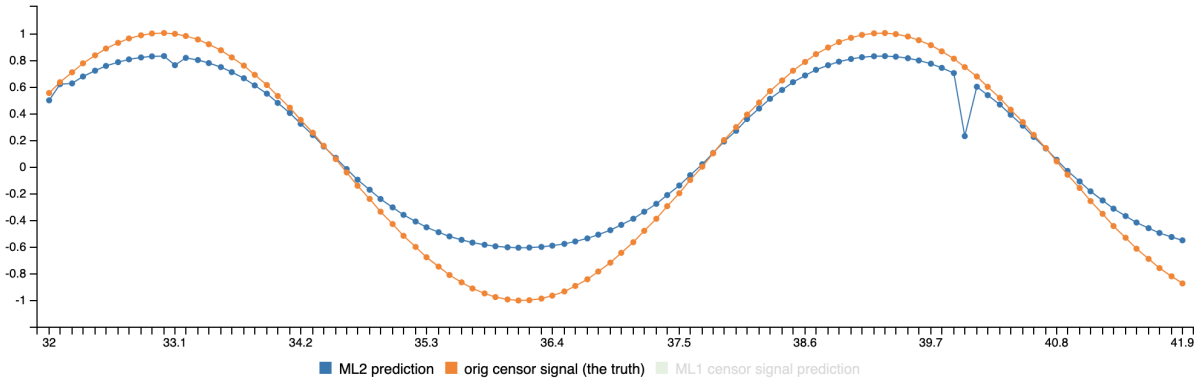
5.2 Testitapaus 1: Tilanne jossa kaikki on mahdollisimman hyvin

Ensimmäisessä tapauksessa molemmat mallit on pyritty opettamaan hyvin. Koneoppimismalli M2 on opetettu ensimmäisen mallin M1 syötteellä. Kuvasta 5.1 nähdään, että malli M1 suoriutuu tehtävästään lähes täydellisesti, se on tulkinnut vain kaksi kuvaa väärin. Kuvasta 5.2 voidaan havaita, että myös malli M2 on melko hyvä, se noudattaa johdonmukaisesti alkuperäisen signaalin muotoa.



Kuva 5.1: Testitapaus 1: Mallin M1 tulkinta signaalista.

Kasvattamalla käytetyn opetusdatan määrää ja opettamalla malli M2 uudelleen voitaisiin todennäköisesti päästä vielä lähemmäs alkuperäistä signaalia. Tämän testiasetelman osalta se ei kuitenkaan ole vaivan arvoista, saamme oleellisen tuloksen esiin vähemmällä ja säästämme samalla laskenta-ajassa. Mallin opettaminen voi olla hyvinkin resursseja kuluttava toimenpide, esimerkiksi ajallisesti pitkäkestoinen. Mallin hyvyys ei ole ainoa kriteeri, mahdollisimman hyvin suoriutuva malli ei ole välttämättä ole kannattavin ratkaisu, jos mallin opettaminen tulee kalliiksi. Tapauskohtaisesti on mietittävät, kuinka paljon resursseja mallin opettamiseen käytetään, ja mikä on sopiva kustannusten ja mallin suorituksen laadun suhde. Mallin kehitystyön alussa olisi hyvä tietää mitä mallilta vaaditaan



Kuva 5.2: Testitapaus 1: Mallin M2 tulkinta mallin M1 tuottaman syötteen pohjalta.

ja miten mitataan se, milloin malli on valmis. Yksi suurimpia haasteita koneoppimismallien kehittämisessä on päätöksen tekeminen siitä, milloin malli on valmis, tarpeeksi hyvä. (Google, 2020)

Tuloksen saamiseen käytetyt pyynnöt:

`http://localhost:8000/sine-prediction/from/31/to/41/freq/1/amp/1/inv/no`
`http://localhost:8000/sine-prediction/front/latest.html`

5.3 Testitapaus 2: Koneoppimismalli M1 on heikko

Tämän koeasetelman lähtötilanne on sellainen, että malli M1 on opetettu heikoksi. Mallin M1 opetus on jätetty tarkoituksella vähäiseksi, jotta saavutetaan tilanne, jossa malli M1 tulkitsee lähes puolet sille näytetyistä kuvista väärin. Testisovelluksen toinen malli M2 on opetettu tämän puutteellisesti opetetun mallin M1 tuottamalla datalla. Testitapauksen seuraavassa vaiheessa opetetaan ensimmäinen malli M1 uudelleen, mahdollisimman hyväksi, mutta jätetään malli M2 ennalleen ja katsotaan miltä tulos näyttää. Lopuksi opetetaan vielä malli M2 uudelleen hyvin suoriutuvan uudelleen opetetun mallin M1 datalla ja todetaan tilanne. Seuraavassa kuvataan tarkemmin, kuinka koeasetelmalla edellä kuvattu asetelma toteutetaan ja lopuksi pohditaan saatuja tuloksia.

5.3.1 Mallin M1 opettaminen heikoksi

Opetetaan ensin malli M1 heikoksi. Muokataan mallin model.ini tiedostoa. Pienennetään [model evaluation metrics threshold values] kohdassa olevaa min accuracy-attribuutin ar-

voa, jotta mallin opetuksen jälkeinen validaatio menisi läpi heikollakin tuloksella. Asetetaan muuttujan min accuracy-arvoksi 0.4. Tämä tarkoittaa, että emme odota mallin tunnistavan kuin noin viidesosan saamistaan näytteistä oikein. Voidaan ajatella, että ensin asetetaan mallin laatua testaavan mittarin arvo halutulle tasolle ja sen jälkeen yritetään opettaa malli vastaamaan tätä laatuvaatimusta. Laaturajoitin ei tosin toimi alaspäin, eli tässä esimerkissä tarkistamme mallin tarkkuuden mallin lokista jonne malli tulostaa tarkkuutensa opetuksen päätyttyä. Opetetaan seuraavaksi mallia M1 tavoitteena saada mallin tarkkuus uuden min accuracy-arvon tasolle. Muokataan mallin hyperparametrien arvoja model.ini -tiedostossa ja kokeillaan millä parametrien arvoilla saavutetaan haluttu tarkkuus. Koneoppimistyönkulkuun kuuluu oleellisesti kokeellisuus ja kokeilu, emme tarkkaan voi tietää millä hyperparametrien arvoilla saavutetaan haluttu tulos mallin opetuksessa. Tämän on yksi syy, minkä takia kehitysympäristön automatisointi ja saatujen tulosten toistettavuus ja jäljitettävyys on haluttua ja tärkeää koneoppimistyönkulussa (Amershi et al., 2019). On hyvä, että jälkikäteen voidaan nähdä, miten jokin tulos on saavutettu.

Muutetaan model.ini -tiedostossa [training config] -osiossa olevien parametrien arvoja. Kokeillaan pienentää opetuskierrosten (epocs) määrää ja katsotaan riittäisikö se halutun tuloksen saamiseksi, muutetaan muuttujan arvoksi 2. Käynnistetään tämän jälkeen mallin M1 opetus projektin juurihakemistossa komennolla python image_recognition.py training. Näemme suorituksen jälkeen mallin tulostamasta lokista, että mallin tarkkuus on vielä liian hyvä MODELS ACCURACY 0.9379. Muutetaan seuraavaksi toisen hyperparametrin arvoa eli muuttujan batch size arvoa. Muutetaan muuttujan batch size arvoksi 4 ja suoritetaan mallin opetus uudestaan. Malli suoriutuu vieläkin liian hyvin. Tässä näkyy koneoppimismallin kehityksen iteratiivinen luonne. Mallin lähdekoodi on valmis, mutta mallin kehitystyö on kesken, malli on vielä opetettava kokeilemalla eri hyperparametrien arvoja ja mahdollisesti myös muokkaamalla opetusdataa.

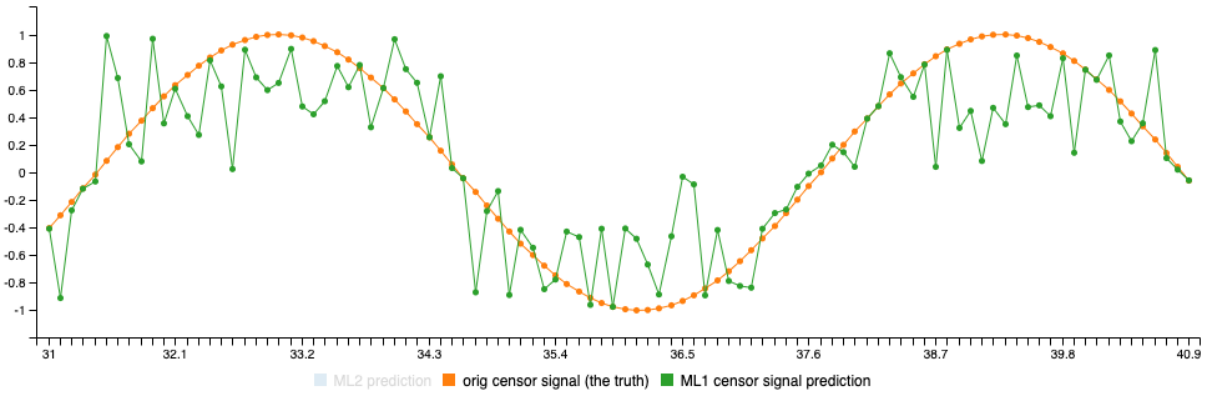
Mallin M1 osalta sen opetusdata on vakioitua, malli M1 käyttää MNIST-kirjaston (LeCun et al., 2020) tarjoamaa opetusdataa, opetusdata haetaan opetuksen alussa Keras-kirjaston kautta. Todellisuudessa hyperparametrien lisäksi myös opetusdata tuo merkittävän muuttujan mallin opetukseen, koska mallista tulee sellainen, millaisella datalla se opetetaan. Voidaan ajatella, että mallin toteuttama funktio ja mallin lähdekoodi muodostaa mallin genotyypin ja data, jolla mallia opetetaan vaikuttaa mallin fenotyyppiin eli siihen millainen malli lopulta on tuotannossa. Kokeilujen jälkeen pääsemme tilanteeseen, jossa mallista tulee liian heikko ja opetus päättyy virheeseen mallin validoinnin kohdalla. Kun epocs hyperparametrin arvo oli 1 ja batch size hyperparamertin arvo 20000 mallin tarkkuudeksi

saatiin 0.3116 mikä on alle model.ini -tiedostossa asettamamme laatukriteerin 0.4. Muuttamalla model.ini -tiedostossa epocs-hyperparametrin arvoksi 2 saamme tuloksen, johon tyydymme, mallin M1 tarkkuus on nyt 0.47150. Karkeasti ottaen malli luokittelee noin puolet sille näytetyistä kuvista oikein.

5.3.2 Mallin M2 opettaminen heikon mallin M1 tarjoamalla datalla

Seuraavaksi opetetaan testisovelluksen koneoppimismalli M2 juuri opetetun mallin M1 tarjoamalla datalla. Seurataan kappaleessa 5.1.2 esitettyjä mallin M2 opetuksessa tarvittavia vaiheita. Mallin M2 hyperparametrien arvot model.ini-tiedostossa pidetään muuttumattomina (batch size=8 ja epocs=25000). Pyydetään tämän jälkeen ennuste ja tulos. Alla esimerkkipyynnöt näihin. Tulos on nähtävissä kuvissa 5.3 ja 5.4.

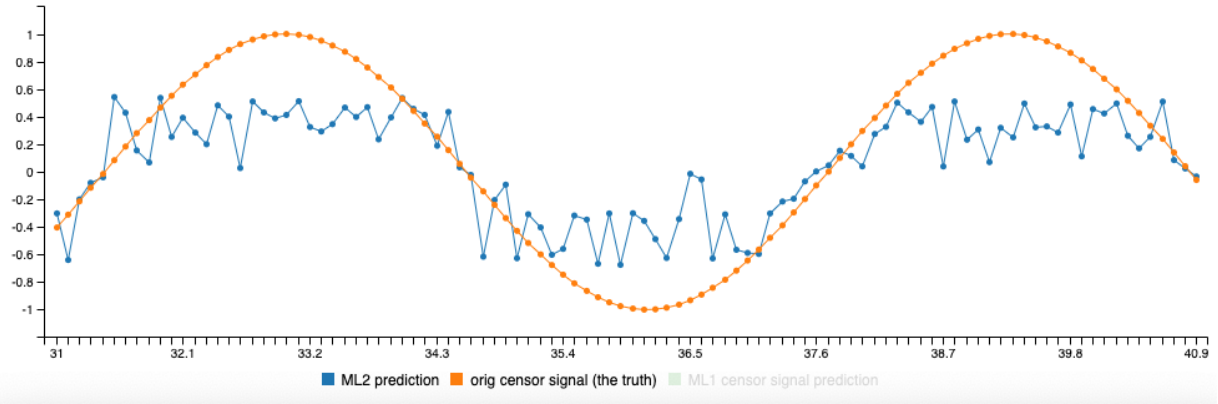
`http://localhost:8000/sine-prediction/from/31/to/41/freq/1/amp/1/inv/no`
`http://localhost:8000/sine-prediction/front/latest.html`



Kuva 5.3: Testitapaus 2: Heikosti tai heikoksi opetettu ensimmäinen malli (M1).

5.3.3 Mallin M1 opettaminen uudelleen mahdollisimman hyväksi

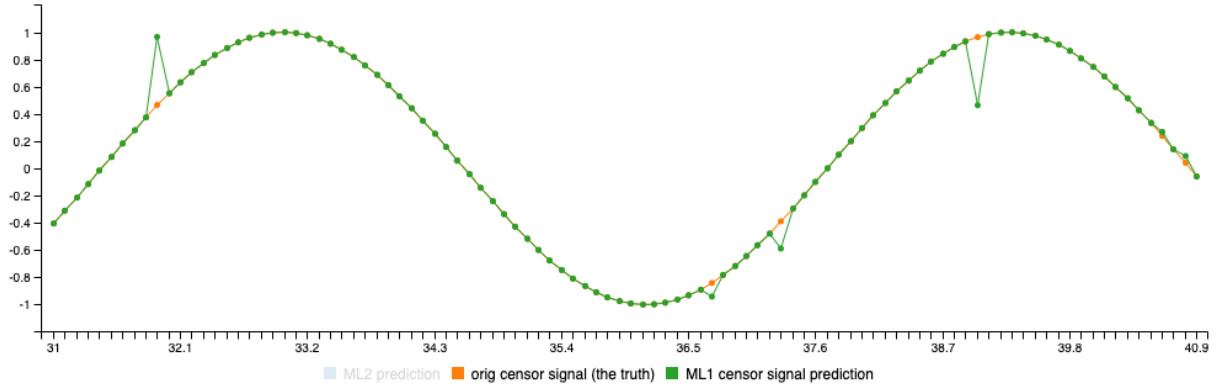
Opetetaan malli M1 nyt uudelleen tavoitteena saada suoritus mahdollisimman hyväksi. Seurataan kappaleessa 5.1.1 esitettyjä askeleita. Mallin M1 hyperparametrien arvot model.ini -tiedostossa ovat seuraavat: batch size=22 ja epocs=11 ja opetuksen validointikriteerin eli muuttujan min accuracy arvo on 0.95.



Kuva 5.4: Testitapaus 2: Heikon ensimmäisen mallin (M1) tuottaman datan pohjalta opetettu malli M2.

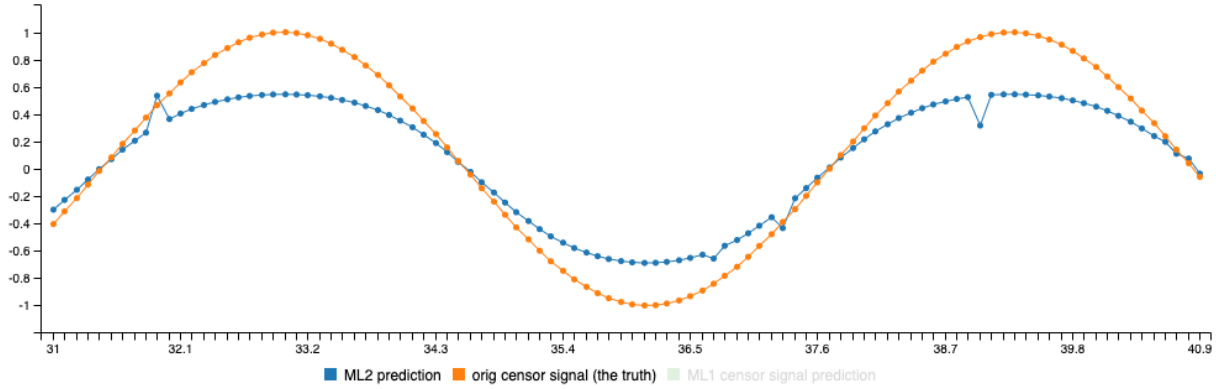
Pyydetään tämän jälkeen ennuste ja tulos. Tulos nähtävissä kuvissa 5.5 ja 5.6. Tuloksen saamiseen käytetyt pyynnöt:

`http://localhost:8000/sine-prediction/from/31/to/41/freq/1/amp/1/inv/no`
`http://localhost:8000/sine-prediction/front/latest.html`



Kuva 5.5: Testitapaus 2: Uudelleen opetettu ensimmäinen malli (M1).

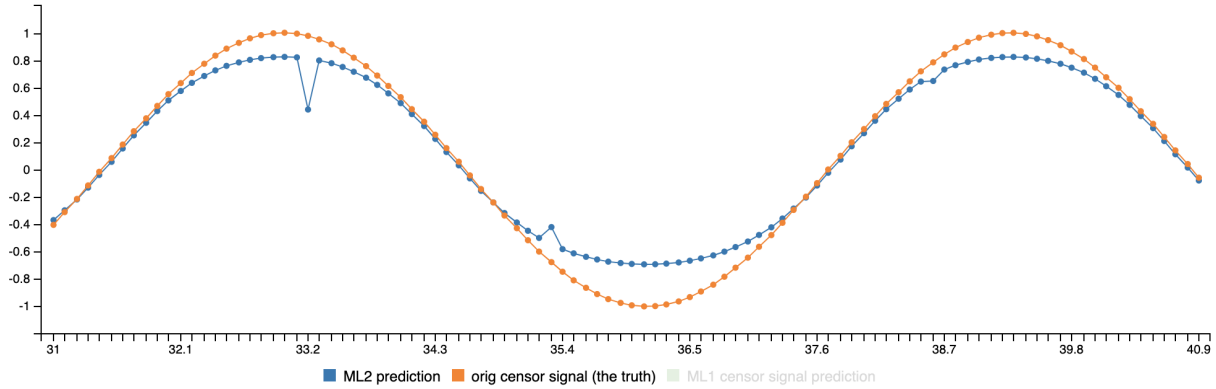
subsectionMallin M2 opettaminen uudelleen hyvin suoriutuvan mallin M1 datalla Opetetaan malli M2 uudestaan uuden mallin M1 tuottaman datan pohjalta. Toimitaan tässä kappaleessa 5.1.2 esitettyjen askelten mukaan. Pidetään mallin M2 hyperparametrien arvot model.ini -tiedostossa muuttumattomina (batch size=8 ja epocs=25000). Otetaan kuvitteellisesti tuotannosta uutta mallin M1 tuottamaa dataa ja käytetään sitä mallin M2 uudelleenopetuksessa opetusdatana. Pyydetään tämän jälkeen ennuste ja tulos. Tämän uudelleenopetuksen tulos on nähtävissä kuvassa 5.7. Nähdään, että mallin M2 tulos parani huomattavasti, se seuraa huomattavasti paremmin todellista sinikäyrää ja tavoittaa melkein myös käyrän huippukohdat.



Kuva 5.6: Testitapaus 2: Malli 2 (M2) uudelleen opetetun ensimmäisen mallin (M1) kanssa.

Tuloksen saamiseen käytetyt pyynnöt:

`http://localhost:8000/sine-prediction/from/31/to/41/freq/1/amp/1/inv/no`
`http://localhost:8000/sine-prediction/front/latest.html`



Kuva 5.7: Testitapaus 2: Malli 2 (M2) uudelleenopetettu mallin M1 uuden datan pohjalta.

5.3.4 Tulosten pohdintaa

Kuvasta 5.3 nähdään, että heikon mallin M1 tuottamassa tulkinnassa signaalista lähes joka toinen numero on tulkittu väärin. Kuvasta 5.4 nähdään, että myös malli M2 on sekaisin tulkinnassaan, koska se on opetettu puutteellisen ensimmäisen mallin (M1) tulkinnan kaltaiseen maailmaan. Mielenkiintoinen kysymys on, voitaisiinko mallin M2 tuloksen pohjalta tehdä johtopäätöksiä opetustarpeesta. Voitaisiinko mallin M2 tuottamasta ennusteesta päätellä tarvitaanko mallien uudelleenopetusta. Kuvasta 5.3 ja kuvasta 5.4 katsoen tämä nähdään selvästi. Kuvista nähdään, että mallien kuvaajat poikkeavat todellisesta

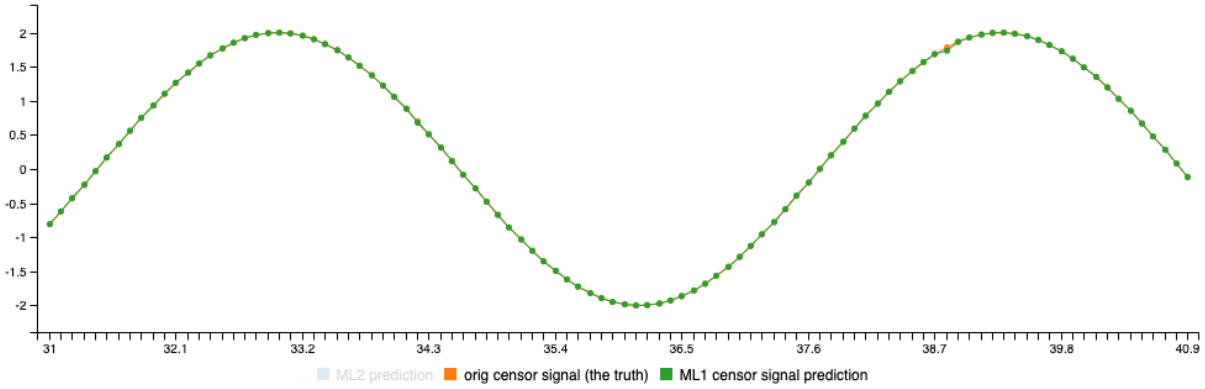
signaalista, mutta voitaisiinko sama päätelmä tehdä ilman kuvaajia. Kuvista nähdään mallien suoritus koko ajanjaksolta kerralla. Voidaan ajatella, että oikeasti nähtäisiin vain yhden ajanhetken tulos kerrallaan ja tiedettäisiin myös jo tuotetut ennustetulosteet. Voisiko yksittäisestä ennusteesta sanoa jotenkin, että malli on hakoteillä? Voisimme yrittää laskea jotain tunnuslukuja, joita käyttämällä pystyttäisiin päättelemään jotain tulosten hyvyydestä. Jos käytössä olisi merkattua (labeled) dataa voitaisiin verrata mallin antamaa tulosta. Koeasetelman simuloimassa reaaliaikaisessa tapauksessa merkattua dataa ei ole suoritushetkellä saatavilla.

Voisimme ajatella, että tunnemme testiasetelmassa mittaamamme ilmiön. Mitattava ilmiö on signaali, joka noudattaa muodoltaan sinikäyrää. Oletuksemme tämän pohjalta on, että ilmiö on luonteeltaan jatkuva ja säännöllinen. Voimme lisäksi päätellä, että peräkkäisten havaintojen ei pitäisi poiketa kovinkaan paljoa toisistaan, koska mitattava ilmiö on jatkuva ja säännöllinen. Peräkkäiset havainnot poikkeaisivat, mutta poikkeaman pitäisi olla suurin piirtein yhtä suuri kahden peräkkäisenä ajanhetkenä tehdyn havainnon välillä. Voisimme asettaa jonkin virhemarginaalin tälle poikkeamalle. Tämä olisi eräänlainen hyvyysluku, jonka puitteissa kahden peräkkäisen mallin M2 antaman ennustetuloksen erotuksen tulisi olla. Mitään suoraa sääntöä tähän ei ole. Koneoppimiselle on luontaista kontekstisidonnaisuus ja empiirisyys, on tunnettava ympäristö, johon mallit kehitetään ja on yritettävä tulkita havaintoja ja dataa siten, että niistä löytyisi säännönmukaisuutta, jonka varaan voisi rakentaa.

Kun malli M1 on opetettu uudelleen hyväksi kuvasta 5.5 voidaan havaita, että mallin suoritus on uudelleenopetuksen jälkeen uudella tasolla, vain kolme tulkintaa sadasta on mennyt väärin. Kuvasta 5.6 nähdään, että malli M2 ei pysty mukautumaan muuttuneeseen maailmaan, ensimmäisen mallin tuottamaan uuteen dataan, vaan tulkinta jää puutteelliseksi. Malli M2 ei tavoita käyrän huippukohtia kovin hyvin. Tämä on looginen ja oletettu tulos, myös malli M2 tulisi opettaa uudelleen, jotta päästäisiin tarkempaan signaalin tulkintaan. Tällaisessa tilanteessa, jossa toiselle mallille syötettä tarjoava malli opetetaan uudelleen, on todennäköistä, että tiedostetaan se, että myös syötteen saava malli voidaan joutua opettamaan uudelleen. Ainakin osataan varautua tähän tilanteeseen ja kiinnittää erityistä huomioita uudelleen opetetun mallin syötettä käyttäviin malleihin. Koeasetelmassa malli M1 opetettiin uudestaan ja sen accuracy-arvo nousi luvusta 0.47150 lukuun 0.95. Tällaisessa tilanteessa, jossa selkeästi nähdään, että mallin suoritus paranee merkittävästi (noin 50 prosentin parannus) on todennäköistä, että myös uudelleenopetetun mallin tuottamaa dataa käyttävät mallit on opetettava uudelleen.

5.4 Testitapaus 3: Signaali muuttuu (signaalin amplitudi kaksinkertaistetaan)

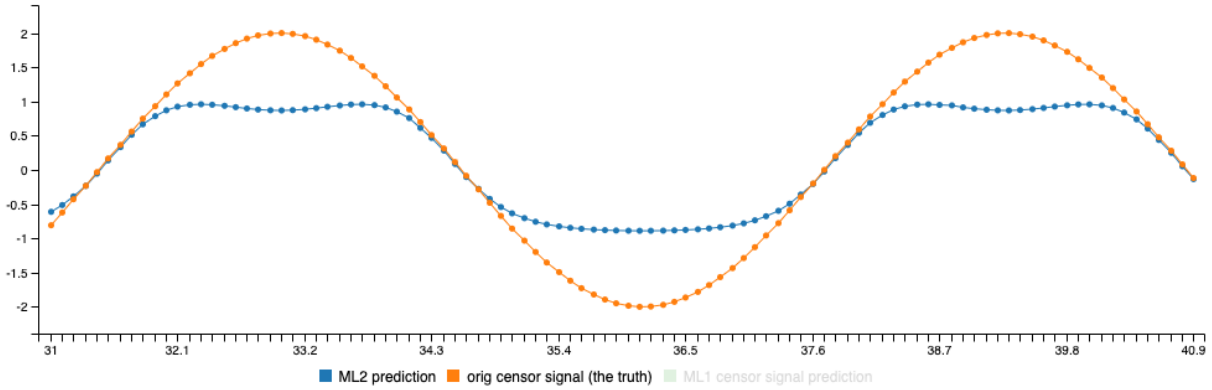
Tässä testitapauksessa pyritään havainnoimaan mikä vaikutus signaalin muutoksella on mallien suoriutumiseen tehtävissään. Lähtötilanne on, että molemmat mallit on opetettu kohtuullisen hyviksi alkuperäisellä signaalilla. Tämän jälkeen mallin M1 syötteenä toimivan signaalin muotoa muutetaan, sen amplitudi muutetaan kaksinkertaiseksi. Kuvasta 5.8 nähdään, että signaalin amplitudi on nyt kaksinkertainen alkuperäiseen nähden. Voidaan myös havaita, että malli M1 on mainiossa vireessä, sadasta tulkinnasta kaksi on mennyt väärin. Mallin M1 suoritus on odotusten mukainen, koska sen kannalta maailma ei ole muuttunut. Malli M1 tulkitsee saamiaan kuvia, se ei ole kiinnostunut tai tietoinen siitä mitä nämä kuvat ja niiden tuottamat numerot kokonaisuutena edustavat, sen tehtävänä on luokitella sille näytettävä kuva kuvaa vastaavaksi numeroksi.



Kuva 5.8: Testitapaus 3: Malli M1 muuttunutta signaalia vasten. (signaalin amplitudi kaksinkertainen opetuksessa käytettyyn verrattuna)

Mallin M2 kohdalla tilanne on toinen. Kuvasta 5.9 havaitaan, että malli M2 on hukassa, voidaan nähdä, että sen tulkinta-alue on opetuksessa käytetty alue lukujen -1 ja 1 välillä, malli M2 ei ole pystynyt mukautumaan signaalin kasvaneeseen amplitudiin. Näyttää, että y:n lähestyessä arvoa 1 mallin M2 tuottama käyrä lähtee laskemaan, vaikka signaalin todellista huippua ei olekaan vielä saavutettu. Tämä näkyy pienenä notkahduksena malli M2 kuvaajassa kohdissa, joissa käyrä lähestyy y:n arvoa yksi. Notkahduksen jälkeen kuvaaja myös pyrkii ylöspäin, kunnes laskee alas. Näyttää, että kuvaaja tavoittelee sinikäyrän säännöllisyyttä, mutta ei saavuta sitä mallin funktion saadessa arvoja, jotka poikkeavat sen kannalta liikaa opetusdatassa käytetyistä arvoista. Jos tarkastellaan kuvaajia lisää, huomataan, että alussa näyttää siltä kuin kaikki olisi niin sanotusti hyvin, mallin M2 en-

nuste noudattaa sinikäyrän mallia. Vasta kun saavutetaan mallin M2 tunteman maailman raja eli mallin opetuksessa käytetyn datan raja eli y:n lähestyessä arvoa 1 nähdään, että mallin M2 tulos romahtaa. Yksi kysymys onkin se missä vaiheessa, miten varhain voidaan havaita, että malli ei suoriudu halutulla tavalla vaan on opetettava uudelleen. Mitä aiemmin havainto uudelleenopetuksen tarpeesta voidaan tehdä sitä parempi.



Kuva 5.9: Testitapaus 3: Malli M2 muuttunutta signaalia vasten. (signaalin amplitudi kaksinkertainen opetuksessa käytettyyn verrattuna)

Voitaisiinko tässä testitapauksessa käyttää testitapauksessa 2 esiteltyä menetelmää mallin hyvyyden tarkkailuun? Testitapauksen 2 yhteydessä pohdittiin voisiko kahden peräkkäisen mallin M2 tuottaman ennusteen erotuksesta päätellä jotain. Voisiko ajanhetkellä n ja ajanhetkellä n+1 tehtyjen ennusteiden erotuksesta päätellä, että malli ei suoriudu tarpeeksi hyvin vaan tarvitsee opettaa uudelleen. Testitapauksen 3 kohdalla vastaus ei ole suoraviivainen, mutta voidaan pitää todennäköisenä, että menetelmä voisi toimia. Riippuu siitä, kuinka suuri virhemarginaali erotuksesta saadulle luvulle annetaan, erotuksen tulos käsitellään itseisarvona, joten etumerkillä ja siten suunnalla ei ole merkitystä. Nähdään, että mallin M2 tuottaman ennusteen kuvaaja on lähes suora niissä kohdissa missä signaali saavuttaa huippukohtansa ja minimikohtansa. Suoralla kahden peräkkäisen ennusteen erotus olisi nolla ja tästä voitaisiin päätellä jotain. Ei kuitenkaan suoraan voida sanoa, että tämä olisi huono tulos, kyseessä saattaa olla vain hetkellinen notkahdus mallin tuloksessa. Olisi ehkä hyvä pitää kirjaa myös peräkkäisten virheiden määrästä tai ylipäätään virheiden määrästä, jotta voitaisiin sulkea satunnaiset notkahdukset tuloksesta pois.

Mikäli peräkkäisten virheiden määrä ylittää tietyn rajan se olisi merkki siitä, että malli suoriutuu heikosti. Peräkkäisten virheiden ja virheiden kokonaismäärän laskeminen ja käyttäminen raja-arvoina mallin hyvyyttä tarkastellessa näyttäisi tulevan merkitykselliseksi siinä tapauksessa, että mallin tuloksessa ei ole suuria heittoja peräkkäisten en-

nusteiden arvoalueiden välillä vaan malli tuottaa tasaiseen tahtiin virheellistä tulosta. Pelkästään peräkkäisten ennustetulosten arvoja vertaamalla ei saada kokonaiskuvaa mallin hyvyydestä, siinä hukataan kokonaiskuva, malli pystyy ikään kuin piiloutumaan tältä mittarilta ja salassa suoriutumaan heikosti. Virheitä olisi hyvä tarkkailla myös suhteessa aikaan. Olisi myös hyvä päästä näkemään virheen suunta, ollaanko etenemässä käyrällä ylöspäin vai alaspäin, onko erotuksen etumerkki positiivinen vai negatiivinen. Suunnan tulkitsemiseksi tarvitsee tuntea aika, olisi tiedettävät millä ajanhetkellä suunnanmuutos on tapahtunut ja kuinka kauan edellisestä suunnanmuutoksesta on kulunut. Huomataan, että tässä lähestytään vähitellen sitä ideaalitilannetta, että käytettävissä olisi merkattua (labeled) dataa jota vastaan ennustetta voitaisiin reaaliajassa tarkastaa, tiedettäisiin ennalta millainen ennusteen on oltava.

Tuloksen saamiseen käytetty pyynnöt:

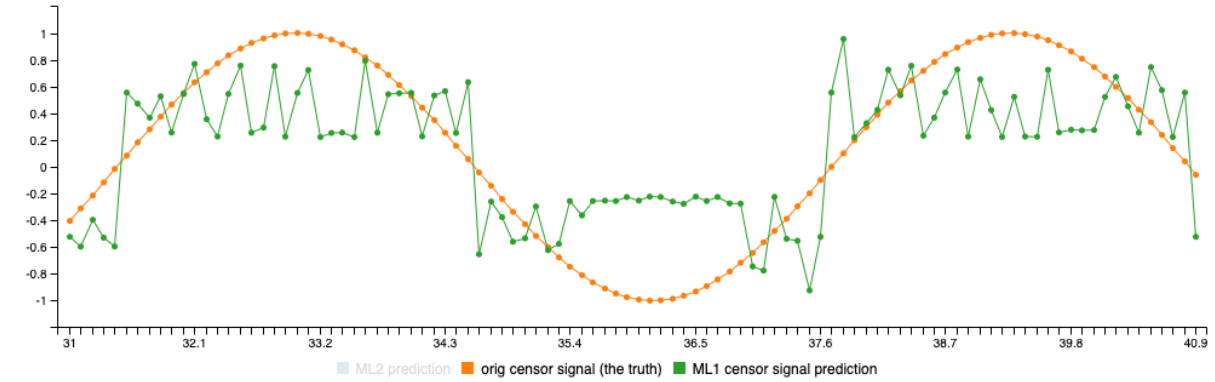
```
http://localhost:8000/sine-prediction/from/31/to/41/freq/1/amp/2/inv/no
http://localhost:8000/sine-prediction/front/latest.html
```

5.5 Testitapaus 4: Signaali muuttuu (MNIST-kuvien värit käännetään)

Neljännessä testiasetelmassa halutaan kohdistaa signaalin muutoksen vaikutus malliin M1 ja havainnoida samalla mikä vaikutus tällä on toiseen malliin M2. Tässä testitapauksessa muutetaan signaalin sisältöä, signaalin koodausta, tapaa, jolla data esitetään. MNIST-kuvat ovat mustavalkoisia kuvia, joissa käsin piirretyt numerot on piirretty mustalla värillä ja kuvien taustaväri on valkoinen. Mallin M1 saama signaali koostuu näistä kuvista ja malli M1 luokittelee saamansa kuvat numeroiksi, jotka malli M2 saa syötteenään. Tässä testissä muutetaan signaalia kääntämällä MNIST-kuvien värit siten, että valkoinen numero on mustalla taustalla. Malli M1 on opetettu datalla, jossa musta numero on valkoisella taustalla.

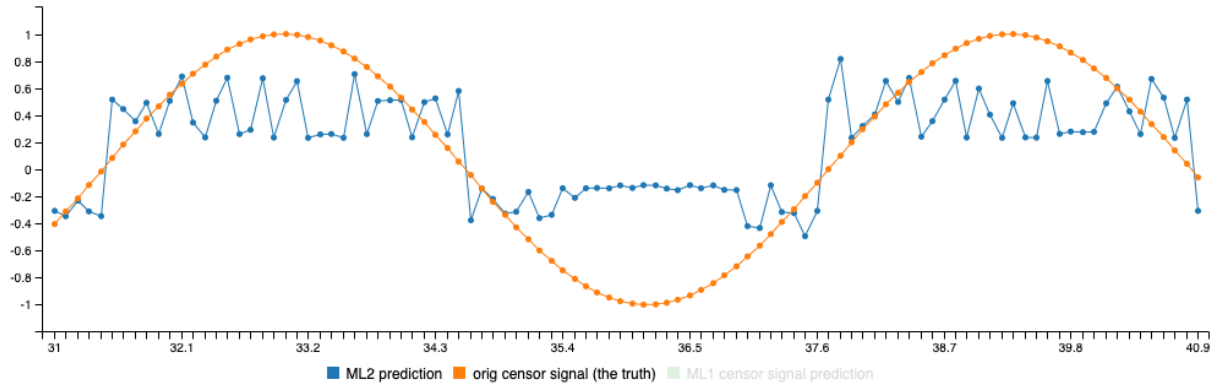
Kuvasta 5.10 nähdään, että mallin M1 tekemä signaalin tulkinta häiriintyy, tulkinnassa on kuitenkin säännöllisyyttä, se noudattelee sinikäyrän muotoa vaikkakin kovin kantikkaasti hukaten sinikäyrän jatkuvuuden. Huomataan, että malli ei pysty tulkitsemaan kovin hyvin kuvia, joissa värit on käännetty. Kuvassa 5.11 on mallin M2 tulos, siinä yksittäiset havainnot heittelevät voimakkaasti, mutta kokonaisuus noudattaa kuitenkin jollain tapaa sinikäyrää, mallin M2 tulkinta on eräänlainen kanttiaalto signaalista. Kuvan 5.12 perus-

teella voidaan havaita, että mallin M1 heikko suoritus näkyy selvästi mallin M2 tuloksessa. Mallin M2 tuottaman sinikäyrän yksittäiset pisteet heittelevät voimakkaasti eivätkä noudata sinikäyrän muotoa.



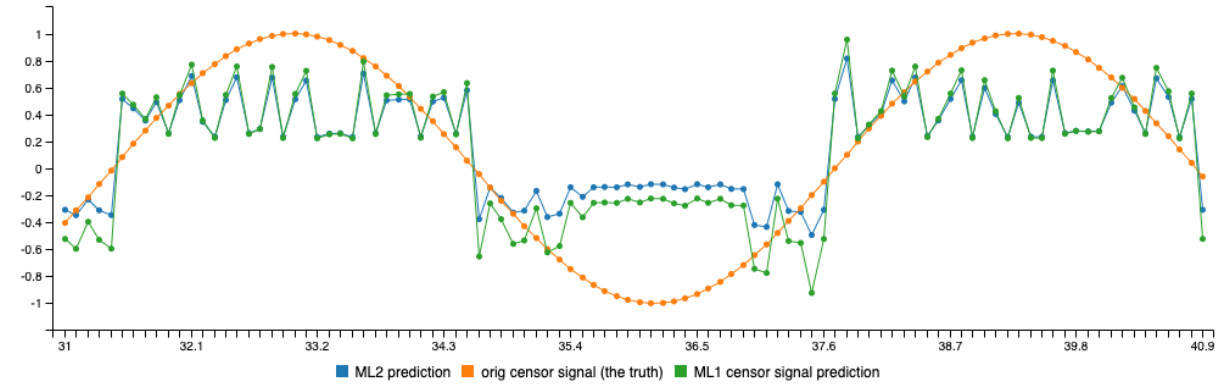
Kuva 5.10: Testitapaus 4: Mallin M1 tulkinta signaalista jossa MNIST-kuvien värit käännetty.

Mallin M2 suorituksen kehnous voitaisiin todennäköisesti havaita käyttämällä samankaltaista keinoa kuin testitapauksessa kaksi esitetty eli laskettaisiin kahden perättäisen ennusteen erotus ja mikäli ennalta määrätty raja-arvo ylittyy se olisi merkki mallin uudelleenopetustarpeesta. Tässä tilanteessa se olisi kuitenkin virhetulkinta, koska mallin M2 opettaminen uudelleen mallin M1 tuotannosta otetulla datalla ei ratkaisisi ongelmaa, jonka perimmäinen syy on muuttunut signaali. Todellinen ongelma on se, että datan rakenne on muuttunut ja poikkeaa siitä datasta millä malli M1 on opetettu. Tämä johtaa siihen, että malli M1 ei pysty tulkitsemaan saamaansa dataa oikein.



Kuva 5.11: Testitapaus 4: Mallin M2 tulkinta ensimmäisen mallin muuttunutta signaalia vasten tekemän tulkinnan pohjalta.

Miten voitaisiin tässä tilanteessa selvittää, että mallia M2 ei tarvitse opettaa uudelleen vaan sen sijaan malli M1 on opetettava uudelleen? Tämä riippunee asetelmasta. Voi olla,



Kuva 5.12: Testitapaus 4: Mallien M1 ja M2 tulkinnat samassa kuvaajassa sinisignaalin kanssa.

että mallien saamaa signaalia tarkkaillaan jollain aikavälillä, jolloin huomattaisiin, että signaali on muuttunut ja tästä pääteltäisiin, että signaalia tulkitseva malli M1 kaipaa uudelleen opetusta. Toinen vaihtoehto on yritys ja erehdys. Kun havaitaan, että mallin M2 suorituksessa on puutteita, päätetään opettaa se uudelleen mallin M1 datalla. Ehkä jo tässä vaiheessa, kun valmistellaan uutta opetusdataa mallille M2 mallin M1 syötteestä, havaittaisiin poikkeamaa mallin M1 tuottamassa datassa ja päädyttäisiin tutkimaan tarkemmin mallin M1 suoritusta ja sen saaman signaalin sisältöä. Mikäli tätä havaintoa ei tehtäisi, opetusdataa valmistellessa havainto tehtäisiin viimeistään mallin M2 opetuksen yhteydessä. Mallin M2 opetuksen yhteydessä todennäköisesti havaittaisiin, että mallin M2 tulos ei parane, vaikka se opetetaan uudelleen tuoreella datalla ja tällöin huomio kääntyisi mallin M1 suoritukseen.

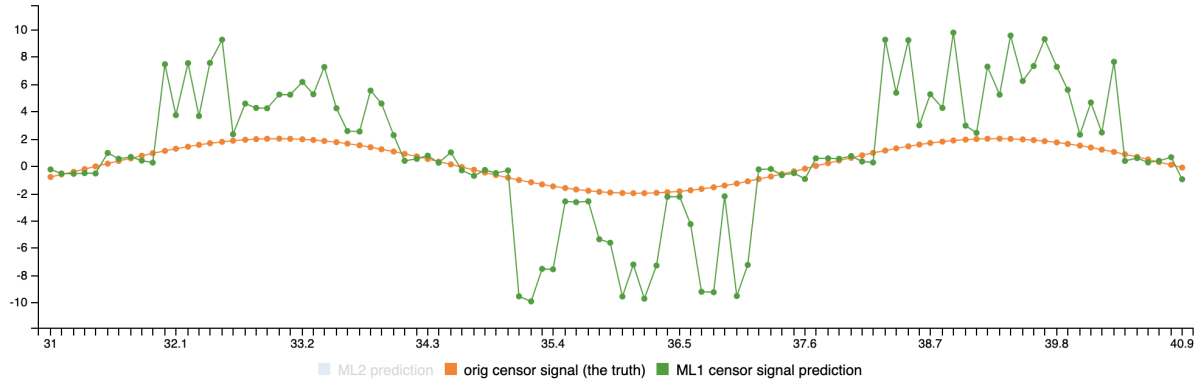
Tuloksen saamiseen käytetty pyynnöt:

```
http://localhost:8000/sine-prediction/from/31/to/41/freq/1/amp/1/inv/true
http://localhost:8000/sine-prediction/front/latest.html
```

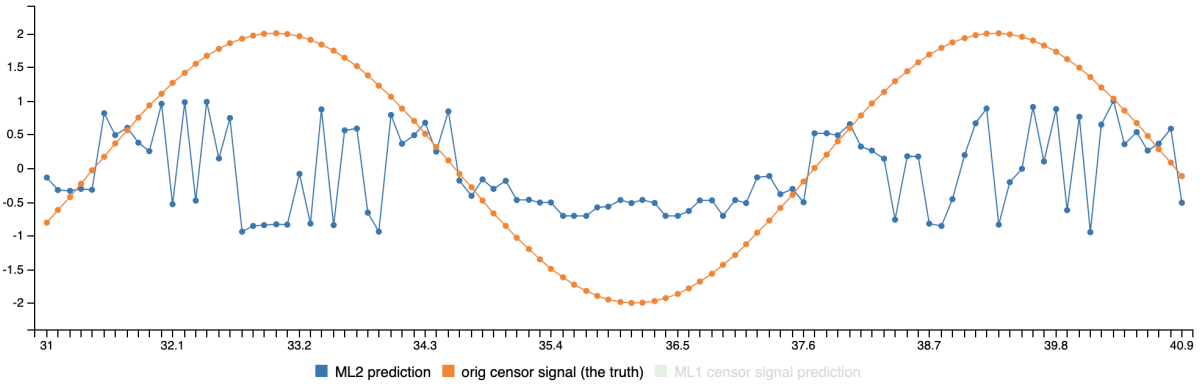
5.6 Testitapaus 5: Signaalin muuttuu (MNIST-kuvien värit käännetään ja signaalin amplitudi kaksinkertaistetaan)

Viidennessä ja viimeisessä testiasetelmassa muutetaan signaalia kahdella tapaa. Yhdistetään testitapauksen 4 ja testitapauksen 3 muunnokset eli käännetään MNIST-kuvien värit negatiiviseksi ja kaksinkertaistetaan signaalin amplitudi. Kuvassa 5.13 nähdään mal-

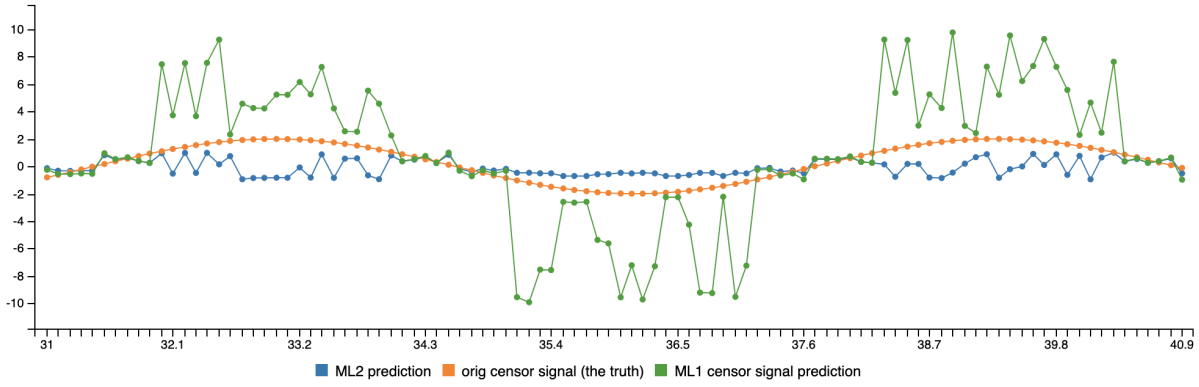
lin M1 tulos ja kuvassa 5.14 mallin M2 tulos. Kuvasta 5.15 mallien M1 ja M2 tulokset rinnakkain.



Kuva 5.13: Testitapaus 5: Mallin M1 tulkinta signaalista.



Kuva 5.14: Testitapaus 5: Mallin M2 tulkinta.



Kuva 5.15: Testitapaus 5: Mallien M1 ja M2 tulkinnat samassa kuvaajassa.

Voitaisiin ajatella, että viidennen testitapauksen tilanne purkautuu kahdessa osassa. Mikäli havainnoidaan mallin M2 kahden peräkkäisen tuloksen erotusta huomataan todennäköisesti poikkeamaa ja tästä päätellään, että malli ei suoriudu halutulla tavalla. Tämä näkyy, kun tarkastellaan mallin M2 ennusteen kokonaiskuvaa 5.14. Voidaan nähdä, että peräkkäisten tulkintojen arvot heittelevät paikoitellen merkittävästi. Jos päädytään siihen johtopäätökseen, että malli M2 kaipaa uudelleenopetusta ollaan samassa tilanteessa kuin aiemmin eli kuinka voidaan tietää tulisiko malli M1 opettaa uudelleen? Olisi hyvä, jos tällaisessa tilanteessa pystyttäisiin varmistamaan datan laatu eli tässä tapauksessa mallin M1 saaman signaalin laatu ja rakenne. Tämä voisi onnistua vertaamalla opetukseen käytetyn datan ja mallin M1 tuotannossa saaman signaalin eroja ja mikäli havaitaan eroja, kerätään uutta opetusdataa ja uudelleenopetetaan malli M1 (Amershi et al., 2019). Tällä tavoin voitaisiin tehdä havaintoja siitä, onko signaalidatan kuvaamassa ilmiössä tapahtunut muutos, onko data muuttunut. Jos havainnoitava ilmiö on muuttunut, voidaan ajatella, että amplitudin muutos kuvaa sitä tässä testitapauksessa, voidaan päätellä, että malli M2 olisi mahdollisesti opetettava uudelleen, mutta mallin M1 opetusta ei välttämättä tarvittaisi. Tämän testitapauksen tilanteessa saatetaan ensin opettaa malli M1 uudestaan, jolloin se osaa tulkita muuttunutta signaalia, eli MNIST-kuvia käännetyin värein. Tämän jälkeen havaitaan, että malli M2 ei edelleenkään suoriudu kunnolla ja viimeistään tässä vaiheessa myös malli M2 opetetaan uudelleen, jotta se osaa mukautua kasvaneeseen amplitudiin.

Tuloksen saamiseen käytetty pyynnöt:

```
http://localhost:8000/sine-prediction/from/31/to/41/freq/1/amp/2/inv/true
http://localhost:8000/sine-prediction/front/latest.html
```

5.7 CI/CD-konfiguraation ja esimerkkisovelluksen rakenteen arviointi

Tässä kappaleessa pohditaan hieman testiasetelman rakennetta ja tehtyjen toteutusvalintojen toimivuutta. Kappaleessa 4, jossa esiteltiin testiasetelman rakennetta, todettiin, että koneoppimismallien integraatiossa on useampia vaihtoehtoja. Tämän työn esimerkkisovelluksessa mallit ovat kiinteästi osa niitä hyödyntävää sovellusta: mallien lähdekoodi ja opetusdata ovat samassa versionhallintajärjestelmässä sovelluksen muun lähdekoodin kanssa. Malleja kutsutaan niitä hyödyntävästä sovelluksesta suoraan niiden metodien kautta. Tämän työn yhteydessä rakenne oli toimiva, mutta käytännössä mallien löyhempi

integrointi niitä käyttävään sovellukseen voisi olla toimivampi ratkaisu. Mallit tarjoaisivat palvelujaan esimerkiksi http-rajapinnan kautta. Tällöin mallit olisivat täysin irrallaan muusta sovelluksesta.

Voidaan ajatella, että tämänkaltainen modulaarisuus, jossa mallien ja muun sovelluksen toiminnallisuus on eriytetty, selkeyttäisi mallien kehitystyötä ja myös niitä käyttävän sovelluksen kehitystä. Mallit voisivat olla eri projektissa ja niillä olisi oma CI/CD-konfiguraatio. Http-rajapinnan kautta mallit voisivat palvella myös useampia sovelluksia yhtä aikaa ilman, että niiden lähdekoodia tarvitsee monistaa useaan projektiin. Malleista voitaisiin tarjoilla myös useita eri versioita http-rajapinnan kautta. Http-kutsun parametrit tai kutsupolku voisi määrätä mikä mallin versio vastaa pyyntöön. Mallit voitaisiin opettaa eri tavalla, tapauskohtaisesti käyttötarpeen mukaan ja kaikki versiot olisivat käytettävissä. Edellä kuvattu rakenne vaatii toki työtä ja voi tuoda lisäkustannuksia, koska malleille pitää olla esimerkiksi oma ajoympäristö minne ne toimitetaan. Http-rajapinnan kautta voi tietyissä tilanteissa tulla myös havaittava viive mallin kutsun yhteydessä. Riippuu sovelluskohteesta minkälaiset viiveet ovat hyväksyttäviä mallin kutsujen yhteydessä. Mallien ollessa kiinteä osa sovellusta tällaista viivettä ei ole. Mallien eriyttäminen omiin projekteihin voisi kuitenkin olla parempi ratkaisu. Mallien versiointi helpottuu ja se mallin versioiden ei tarvitse olla missään yhteydessä niitä käyttävän sovelluksen versioiden kanssa. Myös mallien opetusdata voisi olla osana mallien projektia samassa versionhallinnassa. Toisaalta opetusdatan määrän kasvaessa erillinen säilytyspaikka datalle ja datan monipuolisempi versiointi tulevat ennen pitkää välttämättömäksi.

6 Yhteenveto

Tässä työssä tutkittiin millä tavalla koneoppimismallit vaikuttavat ohjelmistokehitykseen ja tarkemmin CI/CD-käytänteisiin. CI/CD-menetelmät ovat viimeisen kymmenen vuoden ajan tulleet osaksi ohjelmistokehitystä ja on syntynyt DevOps-kulttuuri jatkuvan integroinnin ja toimittamisen ympärille. Jatkuvaan integraation ja toimittamiseen kuuluu vahvasti ajatus ja tavoite ohjelmistokehityksen automatisoinnista. Ohjelmiston lähdekoodi on versionhallintajärjestelmässä jonne ohjelmiston kehittäjät vievät lähdekoodiin tekemänsä muutokset säännöllisesti, muutokset integroidaan näin osaksi kehitettävää sovellusta. Jatkuvan integraation tukena on mahdollisimman pitkälle automatisoidut testit, joiden tarkoituksena on varmistaa, että ohjelmistoon, kehitettävään sovellukseen tehdyt muutokset eivät riko olemassa olevaa toiminnallisuutta. Testit mahdollistavat lähdekoodin jatkuvan integroinnin luomalla rungon muutosten integroimiseksi kehitettävään ohjelmistoon. CI/CD:n toisen puoliskon, jatkuvan toimittamisen, tavoitteena on saattaa jatkuvan integroinnin kautta tehdyt muutokset tuotantoon ja käyttäjien saataville mahdollisimman nopeasti ja hallitusti. Jatkuvan toimittamisen tavoitteena on automatisoida kehitettävän ohjelmiston muutosten toimittaminen tuotantoon. Kehitettävän ohjelmiston ominaisuudet tuovat arvoa vasta kun ne on toimitettu käyttäjille. DevOps on termi, jolla kuvataan sitä millä tavalla ohjelmistoa kehittävä organisaatio on jakautunut kehitettävän ohjelmiston ympärille toteuttamaan jatkuvan integraation ja toimittamisen mukaista toimintamallia. DevOps sulkee sisäänsä kehitystyön, laaduntarkkailun eli testaamisen sekä tuotantoon toimittamisen, ylipäätään kehitystyön mahdollistavan infrastruktuurin ylläpidon ja rakentamisen. Näiden osien kehittäminen ja toteutus voi jakautua eri tavalla riippuen organisaatiosta, kullekin osalle voi olla erikoistuneet tekijät tai sitten henkilöt tekevät useampien osa-alueiden tehtäviä osana työtään. Jatkuva integraatio ja kehittäminen ja niitä tukeva DevOps-kulttuuri pyrkivät automatisoimaan ohjelmistojen kehitystä ja tuomaan ohjelmistokehityksen vaiheille näkyvyyttä ja toistettavuutta.

Ohjelmistokehitys on perinteikkäästi ollut siinä mielessä suoraviivaista, että muutokset ovat kohdistuneet pääasiassa kehitettävän ohjelmiston lähdekoodiin ja kehitettävän ohjelmiston toimintalogiikka on määritelty lähdekoodissa. Tämä suoraviivaisuus on mahdollistanut, tai sen varaan on voitu rakentaa automaattisoituja testejä ja muutokset on ollut verrattain helppoa versioida lähdekoodin mukana versionhallintajärjestelmään. Ohjelmistokehityksessä on nähtävissä muutosta koneoppimismallien lisääntyessä. Koneoppimis-

mallien avulla voidaan toteuttaa ominaisuuksia, jotka aiemmin olisivat olleet vaikeita tai mahdottomia toteuttaa. Koneoppimismalleilla halutaan rikastuttaa ohjelmistoja ja tuoda lisää arvoa ohjelmistojen käyttäjille. Koneoppimismallien mukaantulo kuitenkin monimutkaistaa huomattavasti ohjelmistonkehitystä ja kehitystyön automatisointia (Amershi et al., 2019). Koneoppimismallien tullessa mukaan lähdekoodi ei ole enää ainoa elementti mihin kehitettävän ohjelmiston muutokset kohdistuvat ja joka on versioitava. Koneoppimismallit koostuvat lähdekoodista, opetusdatasta ja opetetusta mallista. Lähdekoodissa määritellään mallin rakenne, sen oppimistapa ja datan hahmottamistapa. Opetusdataa käytetään mallin luomiseen. Opetetun mallin voidaan ajatella edustavan tilaa (state), joka on saavutettu opettamalla lähdekoodin mukainen malli käytössä olevalla opetusdatalla. Opetettu malli edustaa opetusdatan mukaista hetkeä. Mallin opetusta voidaan mukauttaa käyttäen hyperparametrejä, joilla säädetään tapaa, jolla malli oppii. Koneoppimisen myötä versioitavien elementtien määrä kasvaa. Tämä aiheuttaa muutospaineita olemassa oleviin CI/CD-käytänteisiin ja konfiguraatioihin: versioitava on mallin lähdekoodin lisäksi, opetusdata, mallin konfigurointiin käytetyt hyperparametrit ja itse opetuksen tuloksena syntynyt malli. Tässä työssä havaittiin, että mallien versiointi ja opetusdatan hallinta tuovat haasteita CI/CD-systeemiin kun esimerkiksi opetettujen mallien tiedostot on versioitava ja opetusdataa on hallinnoitava. Koneoppimismallit lisäävät vaatimuksia CI/CD-systeemille.

DevOps-termin rinnalle on tullut termi MLOps, joka on päivitetty versio DevOps-mallista ja tuo lisäelementtejä DevOpsin CI/CD-askeleisiin. Koneoppimismallien myötä on pyrittävä lisäämään jäljitettävyyttä ja toistettavuutta. Jäljitettävyyden ja toistettavuuden saavuttamiseksi on tehtävä lisäyksiä käytössä jo oleviin CI/CD-käytänteisiin. Suuret opetusdatamäärät on versioitava, jotta käytetty opetusdata voidaan yhdistää tiettyyn opetetun mallin versioon. Opetetun mallin versioon on yhdistettävä myös opetuksessa käytetyt hyperparametrit ja myös lähdekoodi, jossa malli on toteutettu. Uusien haasteiden ympärille on ilmaantunut palveluntarjoajia ja tuotteita joiden avulla koneoppimiskehitystyöhön ja uusien elementtien versiointiin pyritään saamana toistettavuutta ja jäljitettävyyttä.

Tässä työssä kokeiltiin tehdä GitLab-versionhallintajärjestelmään CI/CD-konfiguraatio tukemaan koneoppimismalleja sisältävän testisovelluksen kehitystyötä. Tämän avulla pyrittiin simuloimaan miten koneoppimismallit vaikuttavat perinteiseen CI/CD-konfiguraatioon. Vaikutukset paljolti riippuvat kehitettävästä ohjelmistosta, siitä millä tavalla koneoppimismallit on otettu käyttöön ohjelmistossa. Ovatko mallit kiinteä osa sovellusta samassa projektissa vai hyödyntääkö ohjelmisto koneoppimismalleja esimerkiksi http-rajapinnan kaut-

ta. Suurimmat haasteet liittyvä mallien opetukseen ja opetusdatan hallintaan. Tämäkin riippuu siitä, minkälainen malli on ja kuinka paljon opetusdataa on. Myös se minkälainen on mallin käyttöympäristö vaikuttaa mallin opetustarpeeseen ja siihen minkälainen CI/CD-konfiguraation tulisi olla tukeakseen mallin elinkaarta, opetustarvetta ja opetusdatan määrää. Mikäli malleja on tarpeen opettaa taajaan, vaikkapa useita kertoja päivässä se asettaa kehitysympäristölle erilaisia vaatimuksia kuin tilanne, jossa mallia opetetaan kerran kuukaudessa (Hazelwood et al., 2018). Opetusdatan hallinta ja datan ja malliversioiden yhdistäminen toisiinsa ovat oleellisia asioita. Tämän työn CI/CD-konfiguraatiossa ja kehitysympäristössä mallin opetusdata tallennettiin samaan versionhallintajärjestelmään muun lähdekoodin kanssa. Opetusdatan määrä oli pieni, joten tällainen ratkaisu oli mahdollinen. Mikäli opetusdatan määrä kasvaisi ja siitä haluttaisiin tehdä esimerkiksi useita versioita niin jonkin muu, mahdollisesti projektin versionhallintajärjestelmän ulkopuolella oleva järjestelmä olisi parempi ratkaisu. Opetusdatan lisäksi mallin opetukseen liittyy mallin hyperparametrit, joita säätämällä mallin opetustulokseen voidaan vaikuttaa. Mallien opettaminen on luonteeltaan pitkälti kokeellista, erilaisia opetusdatan ja hyperparametrien yhdistelmiä kokeillaan parhaan tuloksen saavuttamiseksi (Amershi et al., 2019).

Olisi hyvä, jos CI/CD-konfiguraatio tukisi tällaista kokeellista lähestymistapaa ja mahdollistaisi esimerkiksi opetuksen rinnakkaistamisen käyttäen erilaisia hyperparametreja ja opetusdatakokonaisuuksia. Tämän työn CI/CD-konfiguraation tukee vain yhden opetusaskeleen suorittamista kerrallaan, mutta tätä voisi laajentaa siten että olisi mahdollista suorittaa useampia opetusaskeleita kerralla rinnakkain, jonka jälkeen tuloksia voisi vertailla. Tässä työssä malleille on omat konfiguraatiotiedostot, joissa määritellään hyperparametrit, joita käytetään mallin opetuksen yhteydessä. Yksi jatkokehitysmahdollisuus olisi lisätä mahdollisuus määrittää tiedostoon jokin arvoalue hyperparametrien arvoille ja CI/CD-konfiguraatiossa määritelty opetusaskel automaattisesti suorittaisi opetuksen kaikilla mahdollisilla hyperparametrien yhdistelmillä. Tällöin saataisiin automaattisesti joukko eri parametreillä opetettuja malleja, joiden hyvyyttä voitaisiin vertailla keskenään. Opetuksen jälkeinen mallin hyvyyden päättely on myös kiinnostava asia, tässä työssä mallin hyvyys päätellään mallin tarkkuuden mukaan, opetuksen jälkeen mallilta kysytään sen saavuttama tarkkuus ja tätä arvoa verrataan mallin konfiguraatiotiedostossa annettuun tarkkuuden raja-arvoon. Tätä voitaisiin laajentaa esimerkiksi vertaamalla lisäksi opetetun mallin tulosta mallin edellisen version tulokseen. Mallin konfiguraatiotiedostossa voitaisiin myös määrittää muita laatukriteerejä mallille. Mallin hyvyyden määrittelyssä ja päättelyssä mielenkiintoista on se, että pelkät luvut eivät välttämättä kerro kaikkea. Mikäli malli on ympäristössä, jossa joku toinen malli käyttää sen tulosta syötteenä on

otettava kokonaisuus huomioon ja tutkittava miten yhden mallin näennäisesti parempi yksilösuoritus vaikuttaa koko joukkueeseen. Mallien opetuksessa myös resurssit ratkaisevat, esimerkiksi se kuinka paljon resursseja kuten aikaa on käytettävissä mallin opetukseen.

Toinen tämän työn havainnointikohde oli tilanne, jossa kehitettävässä ohjelmistossa on useampi koneoppimismalli ja mallit ovat toisistaan riippuvaisia. Työssä tehty esimerkkisovellus sisälsi kaksi koneoppimismallia mallin M1 ja mallin M2 ja ne olivat asemoitu siten, että malli M2 käytti mallin M1 tulosta syötteenään. Mallien opetustarpeen havainnointi monimutkaistuu tilanteessa, jossa malli saa syötteensä toisen mallin ennusteesta. Olisi hyvä, jos pystyttäisiin määrittelemään raja-arvoja, joiden perusteella voitaisiin päätellä mallin opetustarve. Mallien suorituksen monitorointi on tärkeässä roolissa, jotta voidaan mahdollisimman aikaisessa vaiheessa havaita poikkeavuudet mallien toiminnassa. Mallien välisten riippuvuuksien monimutkaistuessa mallien suorituksen havainnointi ja opetustarpeen selvittäminen vaikeutuu. Tämän työn testiasetelmassa huomattiin, että jo kahden mallin tapauksessa uudelleen opetuksen päätteleminen monimutkaistuu merkittävästi. Mallien välisten suhteiden ja opetustarpeen monitoroinnissa apua voisi olla mallien välisten viestien monitoroinnista. Seuraamalla mallien välillä kulkevan datan eli viestien sisällön muutoksia voitaisiin mahdollisesti päätellä jotain mallien sisäisestä toiminnasta. Merkittäviä muutos datassa voisi toimia indikaattorina uudelleenopetukselle tai ainakin mallin lähemmälle tarkastelulle (Sethi ja Kantardzic, 2017). Tässä työssä ei toteutettu automaattista mallien valvontaa tai mallien välisen datan havainnointia, mutta nämä olisivat mielenkiintoisia jatkokehitys- ja tutkimuskohteita. Työssä toteutetun testisovelluksen kohdalla voitaisiin esimerkiksi tulostaa lokiin mallien ennustetulokset ja tarkkailla lokiin tulostettujen ennusteiden arvoja, varmistaa että arvot ovat esimerkiksi ennalta asetettujen rajojen sisällä. Voitaisiin myös tarkkailla, poikkeaako mallin syötteenä saama data merkittävästi opetukseen käytetystä datasta. Mikäli tulokset alkavat poikkeamaan raja-arvoista tai mallin saama data etääntyy opetusdatasta, voitaisiin sitä pitää merkkinä mallin ympäristön muutoksesta ja mallin uudelleen opettaminen voisi olla ajankohtaista.

Koneoppimismalleja hyödynnetään laajalta ja käyttö tullee lisääntymään edelleen. Ohjelmistokehityksen työnkulun kannalta mallien opetuksen ja monitoroinnin automatisointi on tärkeää ja suotavaa. Vakiintuneita käytäntöjä ei vielä oikein ole. Se kuinka malleja hyödynnetään ohjelmistoissa, on eräs kiinnostava seikka ja myös se, kuinka mallit on integroitu osaksi ohjelmistoa. Mallit voivat olla kiinteä osa ohjelmistoa osana sovelluksen lähdekoodia toisen ääripään ollessa se, että ohjelmisto käyttää mallin palveluja verkon yli esimerkiksi http-rajapinnan kautta. Tapa, jolla mallit on integroitu osaksi sovellusta

vaikuttaa myös siihen, kuinka mallit olisi hyvä toimittaa tuotantoon. Ovatko mallit omia kokonaisuuksia, jotka toimitetaan tuotantoon erillään muusta sovelluksesta vai ovatko ne kiinteä osa kehitettävää ohjelmistoa. Mallien hyödyntäminen voi arkipäiväistyä jossain vaiheessa jopa niin, että malleja voidaan ottaa ohjelmistoon käyttöön samalla tapaa kuin esimerkiksi NPM-paketteja tai Maven-paketteja JavaScript- ja Java-ohjelmistoissa. Paras tapa ennustaa tulevaisuutta on keksiä se itse kuten Alan Kay aikoinaan totesi.

Kirjallisuus

Algorithmia (2019). *Chaining machine learning models in production with Algorithmia*. [https://algorithmia.com/blog/chaining-machine-learning-models-in-production-with-algorithmia, 6.4.2020].

Amershi, S., Begel, A., Bird, C., DeLine, R., Gall, H., Kamar, E., Nagappan, N., Nushi, B. ja Zimmermann, T. (2019). "Software engineering for machine learning: A case study". Teoksessa: *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. IEEE, s. 291–300.

Bass, Lenn and Weber, Ingo and Zhu, Liming (2015). *DevOps: A Software Architect's Perspective*. Addison-Wesley Professional.

Brown, G. (2010). "Ensemble Learning." *Encyclopedia of Machine Learning* 312.

Brzezinski, D. ja Stefanowski, J. (2013). "Reacting to different types of concept drift: The accuracy updated ensemble algorithm". *IEEE Transactions on Neural Networks and Learning Systems* 25.1, s. 81–94.

CircleCI (2020). *CircleCI*. [https://circleci.com/, 22.3.2020].

Conway, M. E. (2020). *Conway's Law*. [http://www.melconway.com/Home/Conways_Law.html, 21.3.2020].

Dasarathy, B. V. ja Sheela, B. V. (1979). "A composite classifier system design: Concepts and methodology". *Proceedings of the IEEE* 67.5, s. 708–713.

Docker (2020). *Docker*. [https://www.docker.com/, 28.3.2020].

Drone (2020). *Drone*. [https://drone.io/, 22.3.2020].

Dunn, J. (2020). *Introducing FBLearner Flow: Facebook's AI backbone*. [https://engineering.fb.com/ml-applications/introducing-fblearner-flow-facebook-s-ai-backbone/, 27.3.2020].

DVC (2020). *Open-source Version Control System for Machine Learning Projects*. [https://dvc.org/, 27.3.2020].

Ecma International (2017). *ECMA-404, The JSON Data Interchange Syntax*. [http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-404.pdf, 21.3.2020].

Farley, D. ja Humble, J. (2010). *Continuous Delivery: Reliable Software Releases Through Build, Test, and Deployment Automation*. Reading, Massachusetts: Addison-Wesley.

Forsgren, N., Humble, J. ja Kim, G. (2018). *Accelerate: The Science of Lean Software and DevOps: Building and Scaling High Performing Technology Organizations*. IT Revolution Press.

Garcia, R., Sreekanti, V., Yadwadkar, N., Crankshaw, D., Gonzalez, J. E. ja Hellerstein, J. M. (2018). "Context: The missing piece in the machine learning lifecycle". Teoksessa: *KDD CMI Workshop*. Vol. 114.

Garcıa, Á. L. (2019). "DEEPaaS API: a REST API for Machine Learning and Deep Learning models". *Journal of Open Source Software* 4.42, s. 1517.

Garcıa, Á. L., De Lucas, J. M., Antonacci, M., Zu Castell, W., David, M., Hardt, M., Iglesias, L. L., Moltó, G., Plociennik, M., Tran, V. et al. (2020). "A Cloud-Based Framework for Machine Learning Workloads and Applications". *IEEE Access* 8, s. 18681–18692.

Ghanta, S., Subramanian, S., Khermosh, L., Sundararaman, S., Shah, H., Goldberg, Y., Roselli, D. ja Talagala, N. (2019). "Ml health: Fitness tracking for production models". *arXiv preprint arXiv:1902.02808*.

Git (2020). *Git*. [https://git-scm.com/, 22.3.2020].

GitHub (2020). *GitHub*. [https://github.com/, 22.3.2020].

GitLab (2020). *GitLab*. [https://about.gitlab.com/, 22.3.2020].

Gonçalves, E. C., Plastino, A. ja Freitas, A. A. (2015). "Simpler is better: a novel genetic algorithm to induce compact multi-label chain classifiers". Teoksessa: *Proceedings of the 2015 Annual Conference on Genetic and Evolutionary Computation*, s. 559–566.

Google (2020). *Machine learning workflow*. [https://cloud.google.com/ai-platform/docs/ml-solutions-overview, 11.4.2020].

Hazelwood, K., Bird, S., Brooks, D., Chintala, S., Diril, U., Dzhulgakov, D., Fawzy, M., Jia, B., Jia, Y., Kalro, A. et al. (2018). "Applied machine learning at facebook: A datacenter infrastructure perspective". Teoksessa: *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, s. 620–629.

Heroku (2020). *Heroku*. [https://www.heroku.com/, 23.3.2020].

Hutson, M. (2018). *Artificial intelligence faces reproducibility crisis*.

Jenkins (2020). *Jenkins*. [https://jenkins.io/, 22.3.2020].

Kay, Alan (1998). *Alan Kay On Messaging*. [http://wiki.c2.com/?AlanKayOnMessaging, 6.4.2020].

Kay, A. C. (1996). *THE EARLY HISTORY CF SMALLTALK*.

Keras (2020). *Keras: The Python Deep Learning library*. [https://keras.io/, 21.3.2020].

LeCun, Y., Cortes, C. ja Burges, C. J. (2020). *The MNIST database of handwritten digits*. [http://yann.lecun.com/exdb/mnist/, 5.3.2020].

Li, L. E., Chen, E., Hermann, J., Zhang, P. ja Wang, L. (2017). "Scaling machine learning as a service". Teoksessa: *International Conference on Predictive Applications and APIs*, s. 14–29.

Liker, Jeffrey (2004). *The Toyota Way: 14 Management Principles from the World's Greatest Manufacturer*. McGraw-Hill Education.

Liu, S., Feng, L., Wu, J., Hou, G. ja Han, G. (2017). "Concept drift detection for data stream learning based on angle optimized global embedding and principal component analysis in sensor networks". *Computers & Electrical Engineering* 58, s. 327–336.

Miao, H., Li, A., Davis, L. S. ja Deshpande, A. (2017). "Towards unified data and lifecycle management for deep learning". Teoksessa: *2017 IEEE 33rd International Conference on Data Engineering (ICDE)*. IEEE, s. 571–582.

Microsoft (2020). *MLOps: Model management, deployment, and monitoring with Azure Machine Learning*. [https://docs.microsoft.com/en-gb/azure/machine-learning/concept-model-management-and-deployment, 27.3.2020].

MongoDB, I. (2020). *The database for modern applications*. [https://www.mongodb.com/, 21.3.2020].

Pike, R. ja Kernighan, B. (1984). "Program design in the UNIX environment". *AT&T Bell Laboratories Technical Journal* 63.8, s. 1595–1605.

Pinto, F., Sampaio, M. O. ja Bizarro, P. (2019). "Automatic Model Monitoring for Data Streams". *arXiv preprint arXiv:1908.04240*.

Python Software Foundation (2020). *Python programming language*. [https://www.python.org/, 28.3.2020].

Renggli, C., Karlaš, B., Ding, B., Liu, F., Schawinski, K., Wu, W. ja Zhang, C. (2019). "Continuous integration of machine learning models with ease. ml/ci: Towards a rigorous yet practical treatment". *arXiv preprint arXiv:1903.00278*.

Rokach, L. (2010). "Ensemble-based classifiers". *Artificial Intelligence Review* 33.1-2, s. 1–39.

Rosenbaum, Sasha (2020). *CI/CD for Machine Learning*. [https://www.infoq.com/presentations/ci-cd-ml/, 5.3.2020].

Schwaber, Ken and Beedle, Mike (2001). *Agile Software Development with Scrum*. Pearson.

Sculley, D., Holt, G., Golovin, D., Davydov, E., Phillips, T., Ebner, D., Chaudhary, V., Young, M., Crespo, J.-F. ja Dennison, D. (2015). "Hidden technical debt in machi-

ne learning systems". Teoksessa: *Advances in neural information processing systems*, s. 2503–2511.

Sethi, T. S. ja Kantardzic, M. (2017). "On the reliable detection of concept drift from streaming unlabeled data". *Expert Systems with Applications* 82, s. 77–99.

Sonnenburg, S., Braun, M. L., Ong, C. S., Bengio, S., Bottou, L., Holmes, G., LeCun, Y., MÃžller, K.-R., Pereira, F., Rasmussen, C. E. et al. (2007). "The need for open source software in machine learning". *Journal of Machine Learning Research* 8.Oct, s. 2443–2466.

Sridhar, V., Subramanian, S., Arteaga, D., Sundararaman, S., Roselli, D. ja Talagala, N. (2018). "Model governance: Reducing the anarchy of production {ML}". Teoksessa: *2018 {USENIX} Annual Technical Conference ({USENIX}{ATC} 18)*, s. 351–358.

Staples, M., Zhu, L. ja Grundy, J. (2016). "Continuous validation for data analytics systems". Teoksessa: *Proceedings of the 38th International Conference on Software Engineering Companion*, s. 769–772.

Sugimura, P. ja Hartl, F. (2018). "Building a Reproducible Machine Learning Pipeline". *arXiv preprint arXiv:1810.04570*.

Travis (2020). *Travis*. [https://travis-ci.org/, 22.3.2020].

Valohai (2020). *Valohai*. [https://valohai.com/, 28.3.2020].

Wikipedia (2020a). *DevOps*. [https://fi.wikipedia.org/wiki/Devops, 4.3.2020].

– (2020b). *Sine*. [https://en.wikipedia.org/wiki/Sine, 22.3.2020].

Zinkevich, Martin (2020). *Rules of Machine Learning: Best Practices for ML Engineering*. [https://developers.google.com/machine-learning/guides/rules-of-ml, 4.3.2020].

Liite A GitLab-versionhallintajärjestelmän CI/CD-konfiguraatio

Alla .gitlab-ci.yml tiedosto, jolla konfiguroitu testisovelluksen CI/CD-toiminnallisuus GitLab-versionhallintajärjestelmään.

```
# .gitlab-ci.yml
# https://docs.gitlab.com/ee/ci/yaml/
image: python:3.7.1

before_script:
  - echo "Before script section"
  # install dependencies
  - pip install --upgrade pip
  - pip3 install -r requirements.txt
after_script:
  - echo "After script section"
  - echo "For example you might do some cleanup here"

stages:
  - model-training
  - run-unit-tests
  - deploy

unit-tests:
  stage: run-unit-tests
  script:
    - echo "Running unit tests for main.py"
    # this would be the place for some unit tests
  rules:
    - changes:
      - src/main.py

train-ml-1:
```

```
  stage: model-training
  script:
    - echo "Training first model"
    # train the model
    # this throws if the newly trained model does not pass the
    # evaluation
    - python ./src/models/ML1/image_recognition.py training
    # store some info to be used in identifying deployed model and
    # version, this file will be included in deploy,
    - cd src/models/ML1
    - export FILE_NAME="model_training_ci_info.txt"
    - export CI_INFO_FILE="./serialized/$FILE_NAME"
    - touch ./$FILE_NAME
    - echo "JOB_ID=$CI_JOB_ID" > $CI_INFO_FILE
    - echo "JOB_NAME=$CI_JOB_NAME" >> $CI_INFO_FILE
    - echo "PIPELINE_ID=$CI_PIPELINE_ID" >> $CI_INFO_FILE
    - echo "COMMIT_MESSAGE=$CI_COMMIT_MESSAGE" >> $CI_INFO_FILE
    # add model.ini to model_training_ci_info
    - echo "--- model.ini parameters ---" >> $CI_INFO_FILE
    - echo ./model.ini >> $CI_INFO_FILE
    - echo "Saved model training CI info"
    - echo $CI_INFO_FILE
    # if training succeeds upload saved model and training info to
    # Azure cloud
    - python ./../../ci_cd_utils/model_uploader.py ML1 $AZURE_KEY
    # if training succeeds new serialized model is also stored to
    # training job's build artifacts
    # this way we get models versioned
  artifacts:
    name: "ML1-trained-$CI_JOB_NAME-$CI_COMMIT_REF_NAME"
    paths:
      - ./src/models/ML1/serialized/latest_trained_model_1.h5
      - ./src/models/ML1/serialized/model_training_ci_info.txt
  rules:
    - changes:
```

```
        # job is triggered when model's source or configuration changes
        # this model is trained with MNIST data so data changes don't
        # trigger learning
        - src/models/ML1/image_recognition.py
        - src/models/ML1/model.ini

train-ml-2:
  stage: model-training
  script:
    - echo "Training second model"
    # train the model
    # this throws if the newly trained model does not pass the
    # evaluation
    - python ./src/models/ML2/sine_prediction.py training
    - cd src/models/ML2
    - export FILE_NAME="model_training_ci_info.txt"
    - export CI_INFO_FILE="./serialized/$FILE_NAME"
    - touch ./$FILE_NAME
    - echo "JOB_ID=$CI_JOB_ID" > $CI_INFO_FILE
    - echo "JOB_NAME=$CI_JOB_NAME" >> $CI_INFO_FILE
    - echo "PIPELINE_ID=$CI_PIPELINE_ID" >> $CI_INFO_FILE
    - echo "COMMIT_MESSAGE=$CI_COMMIT_MESSAGE" >> $CI_INFO_FILE
    # add model.ini to model_training_ci_info
    - echo "--- model.ini parameters ---" >> $CI_INFO_FILE
    - cat ./model.ini >> $CI_INFO_FILE
    - echo "Saved model training CI info"
    - echo $CI_INFO_FILE
    # if training succeeds upload saved model and training info to
    # Azure cloud
    - python ../../ci_cd_utils/model_uploader.py ML2 $AZURE_KEY
    # if training succeeds new serialized model is also stored to
    # training job's build artifacts
    # this way we get models versioned
  artifacts:
    name: "ML2-trained-$CI_JOB_NAME-$CI_COMMIT_REF_NAME"
```

```
    paths:
      - ./src/models/ML2/serialized/latest_trained_model_2.h5
      - ./src/models/ML2/serialized/model_training_ci_info.txt
  rules:
    - changes:
        # job is triggered when model's source changes
        - src/models/ML2/sine_prediction.py
        # or model's (training) configuration changes
        - src/models/ML2/model.ini
        # or new training data is added to version control
        - src/models/ML2/training_data/training_data.txt
        - src/models/ML2/training_data/training_data_labels.txt

deploy-predictor-app:
  stage: deploy
  script:
    - echo "deploying app to heroku"
    - apt-get update -qy
    - apt-get install -y ruby-dev
    - gem install dpl
    # download saved models and training info (from Azure cloud)
    - python ./src/ci_cd_utils/model_downloader.py ML2 $AZURE_KEY
    - python ./src/ci_cd_utils/model_downloader.py ML1 $AZURE_KEY
    # modify config.js
    - cd src/static_front/js
    - echo "export {PREDICTION_RESULTS_URL as default};" > config.js
    - echo "const PREDICTION_RESULTS_URL = '$PREDICTION_RESULTS_URL';"
        >> config.js
    - cd ../../../
    # add the fetched model files to deploy branch so the files are
    # included in the deploy
    - git config --global user.email "add email"
    - git config --global user.name "add username"
    - git add .
    - git commit -m "add latest models from azure"
```

```
    # do the deploy to Heroku using dpl
    - dpl --provider=heroku --app=gradu-predictor --api-key=$HEROKU_API_KEY
```

Liite B Mallin M1 konfiguraatiotiedosto

```
# https://docs.python.org/3/library/configparser.html
[model info]
model name = latest_trained_model_1.h5
model training name = latest_trained_model_1.h5
description = Some additional, optional info and description
    about the model version.

[training config]
batch size = 22
epocs = 11
verbose = 1
loss function = categorical_crossentropy
metrics = accuracy

[model evaluation metrics threshold values]
min accuracy = 0.95
```

Liite C Mallin M1 lähdekoodi

Alla mallin M1 lähdekooditiedosto image_recognition.py

```
import keras
from keras.datasets import mnist
from keras.layers import Dense
from keras.models import Sequential
from keras.optimizers import SGD
from keras.models import load_model
import argparse
import random
import numpy as np
import configparser

model = None
config = configparser.ConfigParser()
(train_x, train_y), (test_x, test_y) = mnist.load_data()

def transform(number, images, labels, invert):
    """
    Gets number and returns a randomly picked image representing
    the number. If inverted_bw argument is True then invert the
    colors of the returned image.
    """
    img_indices = np.where(labels == number)[0]
    # pick randomly one of the available images
    index = img_indices[random.randint(0, (len(img_indices) - 1))]
    if invert:
        inverted_img = -1. * images[index]
        return inverted_img
    return images[index]
```

```
def get_model_save_path(execution_env, model_name):
    if execution_env == 'gitlab':
        # This path when in CI/CD, training
        return f'./src/models/ML1/serialized/{model_name}'
    else:
        # This path when running with Docker and prod deployment
        return f'./models/ML1/serialized/{model_name}'


def model_satisfies_quality_constraints(accuracy):
    """
    Here we check that the newly trained model satisfies all the
    expectations. For now the only quality metrics is accuracy
    """
    path = './src/models/ML1/'
    config.read(f'{path}model.ini')
    thresholds = config['model_evaluation_metrics_threshold_values']
    return accuracy > float(thresholds['min_accuracy'])


def read_ci_info():
    """
    Read and return model's CI info, the training job's JOB_ID
    in this case.
    """
    ci_filename =
        '/app/src/models/ML1/serialized/model_training_ci_info.txt'
    reader = open(ci_filename)
    try:
        return reader.readline().rstrip().split('=')[1]
    except:
        return 'ERROR - ML1 CI info not read.'
    finally:
        reader.close()


def train_and_test_model():
    path = './src/models/ML1/'
```

```
# read model configuration file
config.read(f'{path}model.ini')
training_conf = config['training_config']
model_info = config['model_info']

# https://keras.io/models/model/
global model
model = Sequential()
model.add(Dense(units=128, activation="relu", input_shape=(784,)))
model.add(Dense(units=128, activation="relu"))
model.add(Dense(units=128, activation="relu"))
model.add(Dense(units=10, activation="softmax"))
model.compile(
    optimizer=SGD(0.001),
    loss=training_conf['loss_function'],
    metrics=[training_conf['metrics']])

# https://keras.io/datasets/#mnist-database-of-handwritten-digits
(train_x, train_y), (test_x, test_y) = mnist.load_data()

train_x = train_x.reshape(60000, 784)
test_x = test_x.reshape(10000, 784)
# https://keras.io/utils/
train_y = keras.utils.to_categorical(train_y, 10)
test_y = keras.utils.to_categorical(test_y, 10)

model.fit(
    train_x,  # train_x[55000:], to test weak model
    train_y,  # train_y[55000:], to test weak model
    batch_size=int(training_conf['batch_size']),
    epochs=int(training_conf['epocs']),
    verbose=int(training_conf['verbose']))

accuracy = model.evaluate(x=test_x, y=test_y,
    batch_size=int(training_conf['batch_size']))
```

```
    if model_satisfies_quality_constraints(accuracy[1]):
        model.save(get_model_save_path('gitlab',
            model_info['model_training_name']))
        return True
    else:
        # raise exception so CI job knows to fail
        raise Exception('Training did not succeed.')

def do_prediction(image_number, invert_img_colors):
    """
    This will be called with an image argument and predicted number
    is returned to the caller.
    """
    path = './models/ML1/'
    config.read(f'{path}model.ini')
    model_info = config['model_info']

    global model
    if model is None:
        model = load_model(get_model_save_path(
            'something_else', model_info['model_name']))
    img = transform(image_number, train_x, train_y, invert_img_colors)
    img = img.reshape((1, 784))
    img_class = model.predict_classes(img)
    prediction = img_class[0]
    return str(prediction)

if __name__ == "__main__":
    parser = argparse.ArgumentParser()
    parser.add_argument("mode", type=str,
       help="The mode the model is executed.")
    args = parser.parse_args()
    if args.mode == 'training':
        train_and_test_model()
```

```
    elif args.mode == 'test_prediction':
        do_prediction()
    else:
        print('ERROR: Unknown mode: {}'.format(args.mode))
```

Liite D Mallin M2 konfiguraatiotiedosto

Alla mallin M2 model.ini konfiguraatiotiedosto

model.ini konfiguraatiotiedosto

```
# config file containing parameters for model training and model's name
[model info]
model name = latest_trained_model_2.h5
model training name = latest_trained_model_2.h5
# Some additional, optional info and description about the model version.
description = [29.3.2020] More epocs!

[training config]
batch size = 8
epocs = 12000
loss function = mean_squared_error
optimizer = SGD
metrics = mean_squared_error
verbose = 1

[model evaluation metrics threshold values]
min accuracy = 0.8
```

Liite E Mallin M2 lähdekoodi

Alla mallin M2 lähdekooditiedosto sine_prediction.py

```
from keras.layers import Activation, Dense
from keras.models import Sequential
from keras.models import load_model
import argparse
import numpy as np
import configparser

DEBUG = True
model = None
config = configparser.ConfigParser()

def transform(signal):
    """
    Transform signal for model
    """
    np_arr = np.array([float(x) for x in signal.split()])
    return np_arr

def get_model_save_path(execution_env, model_name):
    if execution_env == 'gitlab':
        # This path when in CI/CD, training
        return f'./src/models/ML2/serialized/{model_name}'
    else:
        # This path when running with Docker (local dev) and production
        # deployment
        path = './models/ML2/serialized'
        return f'{path}/{model_name}'
```

```
def model_satisfies_quality_constraints(accuracy):
    """
    Read constraints from model.ini file check against here.
    Now returning just true.
    """
    return True


def read_ci_info():
    """
    Read and return model's CI info,
    the training job's JOB_ID in this case
    """
    ci_filename =
        '/app/src/models/ML2/serialized/model_training_ci_info.txt'
    reader = open(ci_filename)
    try:
        return reader.readline().rstrip().split('=')[1]
    except:
        return 'ERROR - ML2 CI info not read.'
    finally:
        reader.close()


def read_training_data_from_file():
    """
    Read training data from filesystem and return the
    data as a numpy array
    """
    file = './src/models/ML2/training_data/training_data.txt'
    with open(file) as f:
        training_data = f.read()
    return np.array(training_data.split()).astype('float')


def read_training_data_labels_from_file():
    """
    Read training data labels from filesystem and return
```

```
    labels as a numpy array
    """
    file = './src/models/ML2/training_data/training_data_labels.txt'
    with open(file) as f:
        training_labels = f.read()
    return np.array(training_labels.split()).astype('float')

def train_and_test_model():
    path = './src/models/ML2'
    # read model configuration file
    config.read(f'{path}/model.ini')
    training_conf = config['training_config']
    model_info = config['model_info']

    train_y = read_training_data_from_file()
    train_x = read_training_data_labels_from_file()
    test_x = train_x[-100:]
    test_y = train_y[-100:]
    train_x = train_x[:-100]
    train_y = train_y[:-100]

    global model
    model = Sequential()
    model.add(Dense(10, input_shape=(1,)))
    model.add(Activation('sigmoid'))
    model.add(Dense(1))

    # configure the learning process
    model.compile(
        loss=training_conf['loss_function'],
        optimizer='SGD',
        metrics=[training_conf['metrics']])

    history = model.fit(train_x, train_y,
              epochs=int(training_conf['epocs']),
```

```
                batch_size=int(training_conf['batch_size']),
                verbose=int(training_conf['verbose']))

    results = model.evaluate(
        x=test_x,
        y=test_y,
        batch_size=int(training_conf['batch_size']))

    if model_satisfies_quality_constraints(results):
        print('Model OK, saving to file system....')
        model.save(get_model_save_path('gitlab',
            model_info['model_training_name']))
        return True
    else:
        print('Model NOT OK, please rerun training')
        # raise exception so CI job knows to fail
        raise Exception('Training did not succeed.')


def do_prediction(signal_data):
    """
    This will be called with signal data
    """
    path = './models/ML2/'
    # read model configuration file
    config.read(f'{path}model.ini')
    model_info = config['model_info']

    global model
    if model is None:
        model = load_model(get_model_save_path(
            'something_else', model_info['model_name']))

    data = transform(signal_data)
    prediction = model.predict(np.array(data))
```

```
    return prediction

if __name__ == "__main__":

    parser = argparse.ArgumentParser()
    parser.add_argument("mode", type=str,
        help="The mode the model is executed.")
    args = parser.parse_args()
    if args.mode == 'training':
        train_and_test_model()
    elif args.mode == 'test_prediction':
        do_prediction()
    else:
        print('ERROR Unknown mode: {}'.format(args.mode))
```

Liite F Pääohjelma

```
#!/usr/bin/env python3
from time import perf_counter
import falcon
import requests
import os
from pymongo import MongoClient
import datetime
from bson import json_util
from falcon_cors import CORS

public_cors = CORS(allow_all_origins=True)

from models.ML1.image_recognition import do_prediction as model_1_pred
from models.ML1.image_recognition import read_ci_info as model_1_ci_info
from models.ML2.sine_prediction import do_prediction as model_2_pred
from models.ML2.sine_prediction import read_ci_info as model_2_ci_info
mongo_client = None
prediction_db = None

try:
    DEBUG = os.environ.get('DEBUG_STATUS', None)
    MONGODB_URI = os.environ.get('MONGODB_URI')
    CENSOR_ADDRESS = os.environ.get('CENSOR_ADDRESS')
    mongo_client = MongoClient(MONGODB_URI)
    prediction_db = mongo_client[os.environ.get('PREDICTION_RESULT_DB')]
except KeyError:
    print('INFO: some env variables not available, using defaults')

def save_prediction_request_results_to_mongodb(prediction_data):
    prediction_collection = prediction_db['prediction_collection']
    result_id =
```

```
        prediction_collection.insert_one(prediction_data).inserted_id
    return result_id

def get_prediction_results_from_mongodb():
    prediction_collection = prediction_db['prediction_collection']
    result = prediction_collection.find()
    json_result = json_util.dumps(result, default=json_util.default)
    return json_result

def get_signal_from_censor(censor_address):
    r = requests.get(censor_address)
    if r.status_code == requests.codes.ok:
        content = r.text
        values = content.split(":")
        return values[0], values[1]
    else:
        return 'no-signal'

def model_1_make_prediction(signal_data, invert_img_colors):
    interpreted_signal = ''
    signal_values_array = signal_data[1:-1].split(', ')

    # MODEL 1 prediction
    for sine_value in signal_values_array:
        past_decimal_point = False
        # possible values to handle:
        # 0.0
        # 0.7833269096274834
        # -0.3738766648302 36
        for char in sine_value:
            if char == '-':
                interpreted_signal += '-'
                continue
            if char == '.':
                interpreted_signal += '.'
```

```
                past_decimal_point = True
                continue
            # this is a special tuning for cases 4 and 5, let
            # the first 0 be 0.
            if char == '0' and past_decimal_point is not True:
                interpreted_signal += '0'
                continue
            interpreted_signal +=
                model_1_pred(int(char), invert_img_colors)
        interpreted_signal += ' '
    return str(interpreted_signal)

def model_2_make_prediction(ml1_signal_prediction):
    interpreted_signal = model_2_pred(ml1_signal_prediction)
    clean_list = list(map(lambda x: float(str(x)[1:-1]),
        interpreted_signal.tolist()))
    return clean_list

def get_prediction(signal_data, time_x, invert_img_colors, **kwargs):
    # get prediction from model 1
    try:
        model_1_signal_prediction = model_1_make_prediction(
            signal_data, invert_img_colors)
        debug_print('model_1_make_prediction done')
    except OSError as e:
        return 'Exception in ML1 prediction: {}'.format(e)

    # get prediction from model 2
    try:
        model_2_signal_prediction = model_2_make_prediction(
            model_1_signal_prediction)
        debug_print('model_2_make_prediction done')
    except OSError as e:
        return 'Exception in ML2 prediction: {}'.format(e)
```

```
    mutated_nums = ''
    first = True
    for key, value in kwargs.items():
        print("%s == %s" % (key, value))
        if first:
            mutated_nums = mutated_nums + value
            mutated_nums = mutated_nums + ' to '
            first = False
        else:
            mutated_nums = mutated_nums + value
            break

    result = {"date": datetime.datetime.utcnow(),
              "censor_original_signal": {
                  "time_line": time_x,
                  "data": signal_data,
                  "signal_alterations": {
                      "invert_img_colors": invert_img_colors,
                      "number_mutation": mutated_nums
                  }},
              "ml1_signal_prediction_result": model_1_signal_prediction,
              "ml2_sine_prediction_result": model_2_signal_prediction,
              "ml1_gitlab_job_artifact_id": model_1_ci_info(),
              "ml2_gitlab_job_artifact_id": model_2_ci_info()}

    mongo_key = save_prediction_request_results_to_mongodb(result)
    result_signal_prediction =
        f'\n ORIGINAL signal: {str(signal_data)}' \
        f'\n time x-axis: {str(time_x)}' \
        '\n\nPredictions:' \
        f'\n\nML1 RESULT: {str(model_1_signal_prediction)}' \
        f'\n\nML2 RESULT: {str(model_2_signal_prediction)}' \
        f'\n\nStored in mongodb with key: {mongo_key}'
    return result_signal_prediction
```

```
class SinePredictionFuture(object):
    cors = public_cors

    def on_get(self, req, resp, start, end, frequency, amplitude, invert):
        signal_req_url = ''
        # if start and end are the same give one
        if start == end:
            signal_req_url = f'{CENSOR_ADDRESS}/single/{start}'
        else:
            signal_req_url =
              f'{CENSOR_ADDRESS}/from/{start}/to/{end}/freq/{frequency}' \
              f'/amp/{amplitude}'
        # get signal
        time_values, signal_data =
            get_signal_from_censor(signal_req_url)
        # should the signal be inverted or not
        invert_signal_images = True if invert == 'true' else None
        result_signal_prediction = get_prediction(
            signal_data, time_values, invert_signal_images)

        resp.status = falcon.HTTP_200
        resp.body = 'Hello from the future! \n' +
            result_signal_prediction

class SinePredictionResults(object):
    cors = public_cors

    def on_get(self, req, resp):
        """
            Get prediction results
        """
        if DEBUG:
            print("SinePredictionResults MAIN - GET - start")
            t0 = perf_counter()
```

```
        results = get_prediction_results_from_mongodb()

        if DEBUG:
            t1 = perf_counter()
            print("SinePredictionResults MAIN - GET end -- Time elapsed: ",
                t1 - t0)

        resp.status = falcon.HTTP_200
        resp.body = results

# falcon.API
app = falcon.API(middleware=[public_cors.middleware])

# resources
prediction_api_future = SinePredictionFuture()
prediction_results = SinePredictionResults()

app.add_route(
    '/sine-prediction/from/{start}/to/{end}' \
    '/freq/{frequency}/amp/{amplitude}/inv/{invert}',
    prediction_api_future)
app.add_route(
    '/sine-prediction/results',
    prediction_results)

app.add_static_route(
    '/sine-prediction/front',
    '/app/src/static_front')
```

Liite G CI/CD apuohjelmat

Mallien lataamiseen pilvipalveluun käytetyt apuohjelmat (model_downloader.py ja model_uploader.py). Näitä kutsutaan GitLab- versionhallintajärjestelmän CI/CD-konfiguraatiosta.

```
from azure.storage.file import FileService
import argparse
"""
Model downloader downloads models from Azure File storage cloud service.
This is called from the deploy job in Gitlab
"""
def download_first_model_and_description_from_azure(key):
    file_service = FileService(account_name='gradumodels',
        account_key=key)
    file_service.get_file_to_path(
        'models',
        'firstmodel',
        'latest_trained_model_1.h5',
        './src/models/ML1/serialized/latest_trained_model_1.h5')
    file_service.get_file_to_path(
        'models',
        'firstmodel',
        'model_training_ci_info.txt',
        './src/models/ML1/serialized/model_training_ci_info.txt')

def download_second_model_and_description_from_azure(key):
    file_service = FileService(account_name='gradumodels',
        account_key=key)
    file_service.get_file_to_path(
        'models',
        'secondmodel',
        'latest_trained_model_2.h5',
```

```
        './src/models/ML2/serialized/latest_trained_model_2.h5')
    file_service.get_file_to_path(
        'models',
        'secondmodel',
        'model_training_ci_info.txt',
        './src/models/ML2/serialized/model_training_ci_info.txt')

if __name__ == "__main__":
    parser = argparse.ArgumentParser()
    parser.add_argument("model", type=str,
                        help="The model that is downloaded. ML1 or ML2")
    parser.add_argument("key", type=str,
                        help="Azure storage key ")
    args = parser.parse_args()
    if args.model == 'ML1':
        download_first_model_and_description_from_azure(args.key)
    elif args.model == 'ML2':
        download_second_model_and_description_from_azure(args.key)
    else:
        print('ERROR Unknown model: {}'.format(args.model))


from azure.storage.file import FileService
from azure.storage.file import ContentSettings
import argparse

"""
Model uploader loads given model to Azure file storage cloud.
This is called from model training jobs in Gitlab
"""
def upload_first_model_and_description_to_azure(key):
    file_service = FileService(account_name='gradumodels',
        account_key=key)
    file_service.create_share('models')
    file_service.create_directory('models', 'firstmodel')
```

```
    file_service.create_file_from_path(
        'models',
        'firstmodel',
        'latest_trained_model_1.h5',
        './serialized/latest_trained_model_1.h5',
        content_settings=ContentSettings(
            content_type='application/octet-stream'))
    file_service.create_file_from_path(
        'models',
        'firstmodel',
        'model_training_ci_info.txt',
        './serialized/model_training_ci_info.txt',
        content_settings=ContentSettings(
            content_type='application/text'))

def upload_second_model_and_description_to_azure(key):
    file_service = FileService(account_name='gradumodels',
        account_key=key)
    file_service.create_share('models')
    file_service.create_directory('models', 'secondmodel')

    file_service.create_file_from_path(
        'models',
        'secondmodel',
        'latest_trained_model_2.h5',
        './serialized/latest_trained_model_2.h5',
        content_settings=ContentSettings(
            content_type='application/octet-stream'))
    file_service.create_file_from_path(
        'models',
        'secondmodel',
        'model_training_ci_info.txt',
        './serialized/model_training_ci_info.txt',
        content_settings=ContentSettings(
```

```
            content_type='application/text'))

if __name__ == "__main__":
    parser = argparse.ArgumentParser()
    parser.add_argument("model", type=str,
                        help="The model that is uploaded. ML1 or ML2")
    parser.add_argument("key", type=str,
                        help="Azure storage key ")
    args = parser.parse_args()

    if args.model == 'ML1':
        upload_first_model_and_description_to_azure(args.key)
    elif args.model == 'ML2':
        upload_second_model_and_description_to_azure(args.key)
    else:
        print('ERROR Unknown model: {}'.format(args.model))
```